1988-06

# Evaluation of work distribution algorithms and hardware topologies in a multi-transputer network

Cloughley, William R.

Monterey, California. Naval Postgraduate School

# NAVAL POSTGRADUATE SCHOOL
## Monterey, California


# THESIS



EVALUATION OF WORK DISTRIBUTION ALGORITHMS AND HARDWARE TOPOLOGIES IN A MULTI-TRANSPUTER NETWORK

by

William R. Cloughley

June 1988

Thesis Advisor: Richard A. Adams

Approved for public release; distribution is unlimited

CURITY CLASSIFICATION OF THIS PAGE

# REPORT DOCUMENTATION PAGE

| | |
|---|---|
| a. REPORT SECURITY CLASSIFICATION<br>Unclassified | 1b RESTRICTIVE MARKINGS |
| a. SECURITY CLASSIFICATION AUTHORITY | 3 DISTRIBUTION / AVAILABILITY OF REPORT<br>Approved for public release;<br>Distribution is unlimited. |
| b. DECLASSIFICATION / DOWNGRADING SCHEDULE | |
| PERFORMING ORGANIZATION REPORT NUMBER(S) | 5 MONITORING ORGANIZATION REPORT NUMBER(S) |
| a. NAME OF PERFORMING ORGANIZATION<br>Naval Postgraduate School | 6b. OFFICE SYMBOL (*If applicable*)<br>Code 52 |
| 7a. NAME OF MONITORING ORGANIZATION<br>Naval Postgraduate School | |
| c. ADDRESS (*City, State, and ZIP Code*)<br>Monterey, CA 93943-5000 | 7b. ADDRESS (*City, State, and ZIP Code*)<br>Monterey, CA 93943-5000 |
| a. NAME OF FUNDING / SPONSORING ORGANIZATION | 8b. OFFICE SYMBOL (*If applicable*) |
| 9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER | |
| c. ADDRESS (*City, State, and ZIP Code*) | 10. SOURCE OF FUNDING NUMBERS |

| PROGRAM ELEMENT NO. | PROJECT NO. | TASK NO | WORK UNIT ACCESSION NO. |
|---|---|---|---|
| | | | |

1. TITLE (*Include Security Classification*)
EVALUATION OF WORK DISTRIBUTION ALGORITHMS AND HARDWARE TOPOLOGIES IN A MULTI-TRANSPUTER NETWORK

2. PERSONAL AUTHOR(S)
Cloughley, William R.

| 3a. TYPE OF REPORT | 13b. TIME COVERED | 14. DATE OF REPORT (*Year, Month, Day*) | 15. PAGE COUNT |
|---|---|---|---|
| Master's Thesis | FROM _____ TO _____ | 1988, JUNE | 97 |

6. SUPPLEMENTARY NOTATION
Approved for public release; distribution is unlimited.

| 7. COSATI CODES | | | 18. SUBJECT TERMS (*Continue on reverse if necessary and identify by block number*) |
|---|---|---|---|
| FIELD | GROUP | SUB-GROUP | Transputer, OCCAM, Multi-Transputer Network, Workfarm. |
| | | | |
| | | | |

9. ABSTRACT (*Continue on reverse if necessary and identify by block number*)

This thesis presents the evaluation of work distribution algorithms and hardware topologies in a multi-Transputer network. The primary emphasis concerns a work distribution algorithm known as "workfarm" that is effective on problems that are divisible into independent work packets.

All the programs and examples presented in this thesis were implemented in the OCCAM programming language, using the Transputer Development System, D700C, Beta 2.0 March 1987 compiler version.

| 20 DISTRIBUTION / AVAILABILITY OF ABSTRACT | 21. ABSTRACT SECURITY CLASSIFICATION |
|---|---|
| ☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS | Unclassified |
| 22a. NAME OF RESPONSIBLE INDIVIDUAL<br>Major Richard A. Adams, USAF | 22b TELEPHONE (*Include Area Code*)<br>(408) 646-2168 |
| | 22c. OFFICE SYMBOL<br>Code 52Ad |

DD FORM 1473, 84 MAR 83 APR edition may be used until exhausted.
All other editions are obsolete

SECURITY CLASSIFICATION OF THIS PAGE

☆ U.S. Government Printing Office: 1986—606-24.

UNCLASSIFIED

Approved for public release; distribution is unlimited.

# Evaluation of Work Distribution Algorithms and Hardware Topologies in a Multi-Transputer Network

by

William R. Cloughley
Lieutenant, United States Navy
B.S., United States Naval Academy, 1981

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 1988

ABSTRACT

This thesis presents the evaluation of work distribution algorithms and hardware topologies in a multi-Transputer network. The primary emphasis concerns a work distribution algorithm known as "workfarm" that is effective on problems that are divisible into independent work packets.

All the programs and examples presented in this thesis were implemented in the OCCAM programming language, using the Transputer Development System, D700C, Beta 2.0 March 1987 compiler version.

## THESIS DISCLAIMER

The reader is cautioned that computer programs developed in this research may not have been exercised for all cases of interest. While every effort has been made, within the time available, to ensure that the programs are free of computational and logic errors, they cannot be considered validated. Any application of these programs without additional verification is at the risk of the user.

Many terms used in this thesis are registered trademarks of commercial products. Rather than attempting to cite each individual occurrence of a trademark, all registered trademarks appearing in this thesis are listed below the firm holding the trademark:

INMOS limited, Bristol, United Kingdom

Transputer

OCCAM

IMS T414

IMS T800

Transputer Development System (TDS)

TABLE OF CONTENTS

DEDICATION

This thesis is dedicated to the following people: to my parents, for having instilled in me the desire to learn and to seek challenge; to LCDR Greg Bryant, whose brilliant thinking and willingness to help lifted the level of learning of the entire Transputer lab; to Major Richard Adams, whose professionalism and selfless dedication to teaching helped me and my classmates immeasurably; and most of all, to my lovely wife Lynnette, who cheered me up when things were not going well and made sure I did not rest on my laurels when I thought they were.

# I. INTRODUCTION

## A. BACKGROUND

The driving force behind modern weapon systems is the processor. At present, most weapon systems utilize a traditional single-CPU serial computer for their processing. Unfortunately, such a computer can process a certain amount of data in a fixed amount of time and no more. A modern radar or sonar can overwhelm the system processor with a flood of raw data. If only part of that data is processed, the rest is lost forever. A new, more powerful computer is required to handle more data. This thesis is part of an effort to improve the processing power of weapon systems.

Multiple computers working in parallel offer significant increases in processing power in a way that provides flexibility and expandability. If one needs more processing power, one adds more computers.

The biggest problem in parallel multi-computer networks is interprocessor communication. This communication is usually handled in one of two ways. The traditional method has been to connect multiple computers together by way of a shared bus. The computers communicate by leaving messages in a single shared memory. The shared memory and shared bus create bottlenecks, however. Memory also creates a bottleneck, since bus bandwidth is higher than memory bandwidth.

The number of computers that can be attached is limited by the bus bandwidth and/or memory bandwidth. A second approach is to have computers communicate with each other by passing messages along direct links. In such systems, each computer has its own memory.

This thesis concentrates on the latter method, since it is attractive for use in a weapon system for reasons of flexibility, growth potential, fault tolerance, lower costs, better response time, and higher system availability.

Microprocessors would seem well suited for a parallel multi-computer network in a weapon system as they are inexpensive, small, lightweight, and increasingly powerful. Until now, however, microprocessors were designed to operate as stand alone computers. Parallelism, with its requirement for communication between computers, was a difficult problem. Since they were not originally intended for this role, their architectures were not suited for it. If they had any provisions for communications at all, they were added on as an afterthought.

A microprocessor family called the Transputer, designed from the ground up for parallelism, is now available from INMOS corporation. It features four full duplex serial communication links and a language also specifically designed for parallel processing. The Transputer is ideal as a building block for a parallel system of microprocessors.

Obviously the evaluation of any computer system, be it single or multi-processor, depends upon the problem. This thesis evaluates a network of Transputers working on a problem that can be broken up into independent work packets. Each packet contains parameters required to process part of the problem. Packets may be combined into a bundle to make communications between nodes more efficient. The amount of computation per packet may be very small or very large and may vary depending on the particular packet.

A critical part of any parallel multi-computer network is the work distribution algorithm. Such an algorithm known as the Workfarm is simple and very effective for many problems including Ray Tracing and Mandelbrot Set and is used in this thesis [MaSh87].

With four links on each Transputer, many physical topologies for a network are possible. To make communication and configuration software less complex, a very regular and symmetric topology is often desirable. There are many such topologies available, from a simple linear array or binary tree to more exotic designs like hypercube or hypernet [HwGh87].

The time it takes for a network of Transputer to complete a problem using a workfarm is primarily dependent on the following factors: the number and speed of the Transputers, the number of computations per packet, the number of

packets per bundle, and the total number of packets in the problem.

This thesis will examine how these factors are interrelated. The two primary questions addressed are, given a workfarm, 1) how many Transputers will be required to solve a problem given a time limit, and 2) how fast can a problem be solved for a given number of Transputers?

The resulting predictions will be compared to actual results on two specific problems: Mandelbrot Set [Po86] and Coordinate Transformation [Ri87]. These problems can be characterized by their divisibility into work packets which can be processed in any order and by their massive computational requirements.

## B. THESIS OVERVIEW

Chapter II presents a brief look at the Transputer system. Chapter III discusses the workfarm, how it is implemented on a network of Transputers, and what topology is used. Chapter IV presents the results of timing studies using the workfarm. Chapter V uses the findings of Chapter IV to predict the performance of the Coordinate Transformation and Mandelbrot Set problems and compares them with the actual results. Chapter VI presents the conclusions and recommendations of the thesis.

# II. THE SYSTEM

## A. GENERAL

It is hard to separate the Transputer hardware and Occam software. They are so tightly intertwined that is easier to treat them as a single entity; the Transputer system. A basic understanding of this system is necessary when programming a network of Transputers. Although this learning may seem onerous, it may well be the reason Transputers are relatively easy to program in parallel compared with other microprocessors and languages. The reader is assumed to be familiar with the fundamentals of the Transputer architecture and the Occam programming language. For those unfamiliar with Transputers, [In88] and [PoMa87] are excellent starting points. This chapter highlights the knowledge necessary for efficient parallel processing and performance maximization.

Despite a network of Transputers's tremendous power, a program has to be as efficient as possible to realize the full potential. To do so, the programmer must first ensure that each individual Transputer is optimized and then that the network is. The latter is primarily a matter of work distribution and communication. The following is a synopsis of Transputer performance maximization; a complete reference can be found in [At87].

## B. SINGLE TRANSPUTER OPTIMIZATION

It is extremely desirable to keep all the data structures and code in the on-chip memory. An on-chip memory cycle takes one processor cycle (50 nanoseconds for the latest 20MHz Transputers). Although variable, depending on the instruction and on the speed of the DRAM chips, external memory references usually take five processor cycles [In87a]. In other words, one external memory access takes five times as long as an on-chip memory access.

Given a choice between data structures on-chip or program code on-chip but not both, one would choose data structures. A Transputer word holds four bytes; one memory access of program code returns four single byte instructions. That, combined with the principle of locality, means accessing program instructions from external memory is not nearly as slow as accessing data structures from external memory.

Transputer memory is arranged as follows, from the base of memory upwards: system space, data space, code space, with any unused space on top. Off chip (external) memory is not allotted until there is no free on-chip memory remaining. The Occam compiler and Transputer loader software automatically places data (data structures, workspaces, etc.) lower than program code on the Occam map [At87]; hence, program code only resides in on-chip memory if the data space has already been accommodated.

The programmer has control over the order in which data structures occur in memory through the order in which they are declared in the code. Simply put, data structures declared last in a local block are placed lower in the memory map than their predecessors.

Large data structures, such as arrays, should be declared first in a local block, followed by any variables, so that the variables are allotted memory lower in the map, ahead of the large arrays. Otherwise, frequently used variables may be allotted off chip memory. To ensure large data structures do not monopolize on-chip memory, they can be declared in a global process. In keeping with proper programming practices, i.e., declaring data structures locally, the global data structures can be artificially declared locally using abbreviations with no performance cost.

Concurrent process workspaces also reside in the data portion of the memory map. The programmer controls which processes lie lower in the memory map by the order in which they are declared. Like data structures, those processes declared last are allotted workspaces lower in the memory map.

To finetune the Transputer further, awareness of where variables lie relative to the workspace pointer is useful. Only a single byte instruction is required to manipulate the first 16 locations above the workspace pointer because the four bit relative address fits in the lower half of the

single byte instruction. This optimization is technique is useful in a local block with more than 16 variables.

## C. MULTIPLE TRANSPUTER OPTIMIZATION

A key to maximizing the performance of a network of Transputers is decoupling communication from calculation. This is accomplished by running communication and calculation processes separately and in parallel. Other processes, also running in parallel, act as buffers between the communication and calculation processes. The processes communicate among themselves through internal or external channels. With such a setup, a Transputer may now communicate and calculate simultaneously, with little degradation in either area.

Equally important, the communication processes must always run at high priority and the calculation processes at low priority. Consider a network of Transputers. If each Transputer only passes messages when finished its own calculation, the majority of the Transputers in the network will constantly lie idle, waiting to receive work or to send results. The communication must take precedence so that the message passing is uninhibited.

Because the communication and calculation are decoupled in each Transputer, the communication does not significantly affect the ongoing calculation. How much calculation degradation actually occurs during ongoing communication was researched in [Ha87] and [Br88]. In a nutshell, a single

Transputer communicating on all four links at full capacity, which is the worst case scenario, can still calculate at approximately 75% of its maximum capability [Ha87]. The separate DMA engines on the chip make this possible, although the degradation is caused by internal bus contention between the link engines and the central processor.

The actual link data rates have been investigated previously in [Va87] and [Br88]. To summarize, one can expect rates of approximately 2.3 Mbytes/second through T800 links during bidirectional communication and 1.7 Mbytes/second during unidirectional communication. The T414 has rates of 1.5 Mbytes/second during bidirectional communication and 0.76 Mbytes/second during unidirectional communication [Br88]. These rates assume no external memory usage. The T800 communicates significantly faster than the T414 because of a handshaking improvement in link communication [In87b].

A third area in which a programmer can significantly affect performance is the length of the communication message. The overhead to send a single integer over a link is the same as that to send an array of 100 integers, for example. Obviously it is better to keep messages as long as possible and cut down on the overhead.

As the message length grows, however, the probability increases that its data structure will reside in off chip memory. That of course means significant performance degradation.

The trick is to keep the message as long as possible without going into off chip memory. The programmer can figure out the happy medium by keeping track of how much data structure space he is using and making sure it is less than the on-chip memory size. If it is more, the programmer has to shorten his message arrays until all the data structures fit in on-chip memory.

D. MISCELLANEOUS TIPS

As pointed out in [Br88], the timer should not be utilized in the B004 T414 Transputer. It does not return an accurate measure of time because the T414 is executing processes that the user is not aware of and these hidden processes figure into any timing measurements. The timer should be utilized on a remote Transputer and the time returned over a link for display.

Occam is very strongly typed. In a network of Transputers, where message passing is paramount, this comes heavily into play. The type of data at one end of a channel must be the same type that comes out the other end or the program deadlocks. This error is particularly insidious because there is always a lot of communication in a network and there are no messages to tell why the program has stopped. For this reason, it is imperative to begin testing on a B004, followed by testing on a single remote Transputer, before testing a program on a network of Transputers.

Protocols are used to identify what is coming across a channel; however, only data of that protocol will be able to use the channel. A more flexible approach is use of the CHAN OF ANY declaration which allows the programmer to send anything down a channel as long as the same thing comes out the other end.

Variant protocols offer the same flexibility. They are expensive in terms of overhead, however. A cheaper but trickier method is to declare channels using CHAN OF ANY and sending arrays through these channels with a single byte or integer tag at the head. The tag allows the receiver to know what type of data follows in the array. This manual method requires retyping, a practice requiring much care from the programmer, and should be well tested on a single remote Transputer before it is used in a network.

All Transputer links may run at the standard speed of 10 Mbits/second; however, most members of the Transputer family are now capable of running their links at 20 Mbits/-second. The B004, B002, and B001 are the only commonly used boards restricted to the standard link speed. Fortunately, the 20 Mbits/second capable boards allow their Link 0s to be set at 10 Mbits/second while the rest of the links run at 20 Mbits/second. Normally a network of Transputers is run at 20 Mbits/second for fastest communications except where an input is accepted from a B004 host computer or B001/2 RS232 interface board. It is easy to tell if the DIP switches have

been improperly set; an error message will flash when the extracted code will not load through the mismatched link.

## E. DEBUGGING

The Transputer's internal architecture time slices parallel processes to simulate parallel processing. This aids the programmer because he can test his program on one Transputer; then, with minimal change, run his program on a network of Transputers as intended.

Based on programming experiences, it is recommended that a program intended for a network of Transputers be developed in three steps. First, the calculation process is tested sequentially on the B004 T414 Transputer. The B004's host computer allows the programmer to easily display any variables in the executing program.

The next step is to run the same program on a remote Transputer connected to the B004. Any bugs with the external links or channel protocols will reveal themselves in this step. Because a program may work on one Transputer using internal channels but not on multiple Transputers using external channels, this step is important. During execution, variables may be passed over external links back to the B004 for display.

Finally, the program is run on a network of Transputers. There may still be bugs but at least the programmer knows that a large portion of his code is good, especially the channel protocols. The first two testing systems remain and

the programmer can reuse them to check out any questions he may have.

The problem is that in a network of Transputers a program either works or it deadlocks. When a program deadlocks, all the programmer sees is a blinking cursor. There is very little the programmer can do to determine what went wrong, where the problem occurred, or to obtain a state trace. Additionally, because the Transputers run in parallel, it is next to impossible to display variables to a monitor during execution.

# III. THE WORKFARM

## A. GENERAL

A problem has to be divisible for use in a parallel system. A parallel system of Transputers is so powerful that the problem should also be computation intensive. This thesis deals with problems that are both divisible and computation intensive. Problems that are not divisible (assuming a single input data stream) are not germane to this thesis.

## B. PIPELINE

The pipeline is a well known work distribution algorithm. It is ideal for problems that divide into tasks that can be assigned to separate processors. With the Occam programming language and the Transputer links, it is easy to configure a network of Transputers into a pipeline, as demonstrated in Figures 3.1 and 3.2.

Unfortunately, pipelines are only as fast as their slowest process. In a pipeline, the Transputers execute the following cycle:

```
WHILE TRUE
  SEQ
    communicate
    synchronize
    calculate
    synchronize.
```

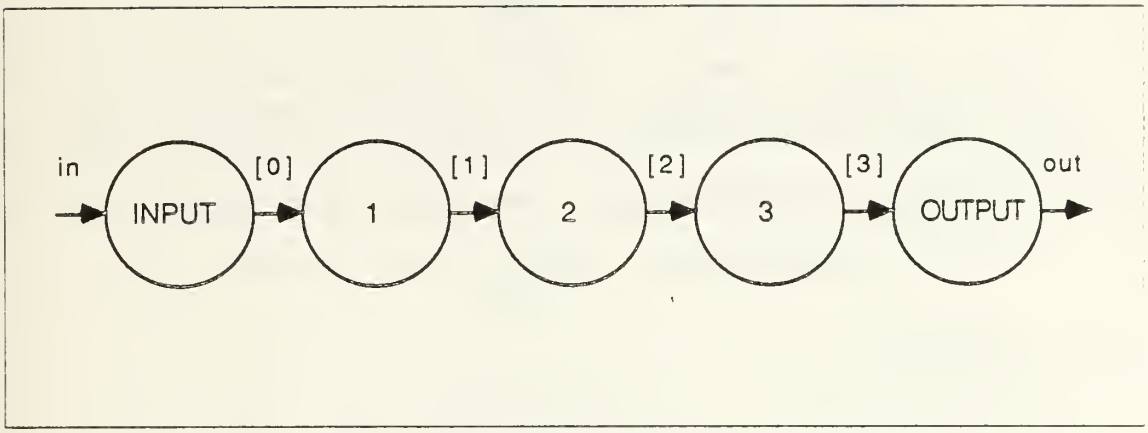


Figure 3.1

Transputer Pipeline

```
[4]CHAN OF ANY  chan:
PAR
  input  (in,       chan[0])
  proc1  (chan[0], chan[1])
  proc2  (chan[1], chan[2])
  proc3  (chan[2], chan[3])
  output (chan[3], out)
```

Figure 3.2

Transputer Pipeline Code

To maximize performance, the calculation time of the processes must be equal. This limits the problem range.

A special case is when each processor in a pipeline can do the same task. Heat transfer along a wire is such a problem, although in this case the pipeline is bidirectional. The calculation time for each processor is equal and so the network can achieve peak efficiency. The configuration for this special pipeline, as demonstrated in Figures 3.3 and 3.4, is even simpler than that of the standard pipeline.

C. WORKFARM

The workfarm is a work distribution algorithm in which each processor does the same task on part of a problem instead of each processor doing a separate task as in the standard pipeline. It is highly effective on problems that can be divided up into independent work packets, where each packet consists of parameters necessary to calculate a part of the problem. Independent means that no packet is dependent on any others. If one packet of the total were lost, only a small piece of the problem would be missing. The amount of work required per packet may vary.

The workfarm has two distinct parts: the Controller and the Farm. The controller combines packets into request bundles and passes the request bundles to the farm, ensuring that there are never more bundles in the farm than the farm can handle. The controller also receives the result bundles.

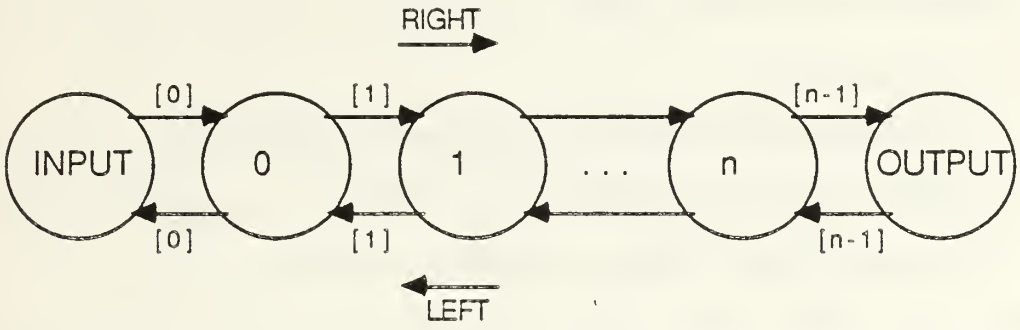


Figure 3.3

Bidirectional Pipeline

```
[num.nodes+1]CHAN OF ANY right, left:
SEQ
  input (right[0], left[0])
  PAR i = 0 FOR num.nodes
    node (right[i], right[i+1],
          left[i+1], left[i])
  output (right[num.nodes], left[num.nodes])
```

Figure 3.4

Bidirectional Pipeline Code

The packets are grouped into bundles to optimize the length of the message arrays as discussed earlier.

The farm consists of multiple nodes arranged in some topology. This thesis uses a linear array of nodes for reasons explained later. A workfarm of this type is pictured in Figure 3.5.

When a node receives a request bundle and is not "busy" it accepts the new one for processing. However, if the node is busy when the request bundle arrives, it passes the bundle to the next node (further away from the controller). Result bundles, arriving from the opposite direction, are simply passed along to the next node until they reach the controller. When a node finishes processing a bundle, it sends its result bundle towards the controller.

In the workfarm, each node has the same code, although sometimes it may be desirable to make minor modifications to the end node code.

The configuration code for a workfarm is listed in Figure 3.6. Two arrays of channels are declared, one to carry the request bundles out to the farm and the other to return the result bundles back to the controller. Expanding the farm of Transputers is as easy as changing the constant 'num.Transputers'. Note that the controller and node processes each have been "placed" on a separate Transputer, in this case a T414 controller and T800 nodes.

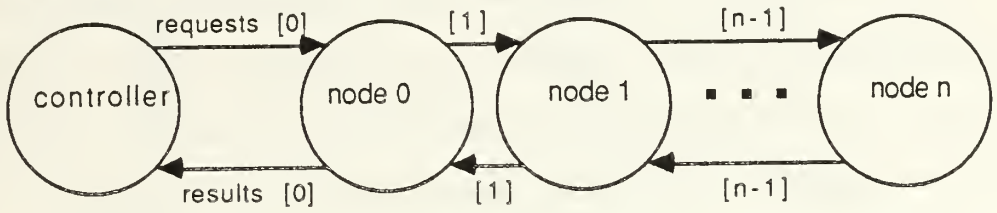


Figure 3.5

Linear Array Workfarm

```
[num.Transputers]CHAN OF ANY requests, results:
PLACED PAR
  PROCESSOR T4 100
    controller (requests[0], results[0])
  PLACED PAR i = 0 FOR num.Transputers
    PROCESSOR T8 i
      node (requests[i], requests[i+1],
            results[i+1], results[i])
```

Figure 3.6

Linear Array Workfarm Code

The controller consists of four processes running in parallel as shown in Figures 3.7 and 3.8. Requests and results are external channels passed in by the global configuration process; all other channels are internal to the controller and must be declared.

The generator initiates the entire workfarm process by creating the request bundles and passing them to the work router. If necessary, the generator may receive inputs from outside sources through external channels to build the packets and bundles. The generator signals the handler just before it begins to pass out bundles and is in turn signalled by the handler when the handler has received the last result bundle.

The work router and result router are essentially buffer processes. Together, they also perform a vital valve function to make sure the farm never exceeds its capacity for request bundles. The work router knows how many nodes are in the farm. Since each node has a buffer enabling it to hold two bundles at a time, the work router knows the farm can hold twice as many request bundles as the number of nodes. It was necessary to reduce by one the farm bundle capacity known to the work router to avoid overloading the farm.

When the work router passes a bundle to the farm, it increments a counter. When the results router receives a bundle, it signals the work router through the trigger channel. When the work router is signaled, it decrements the

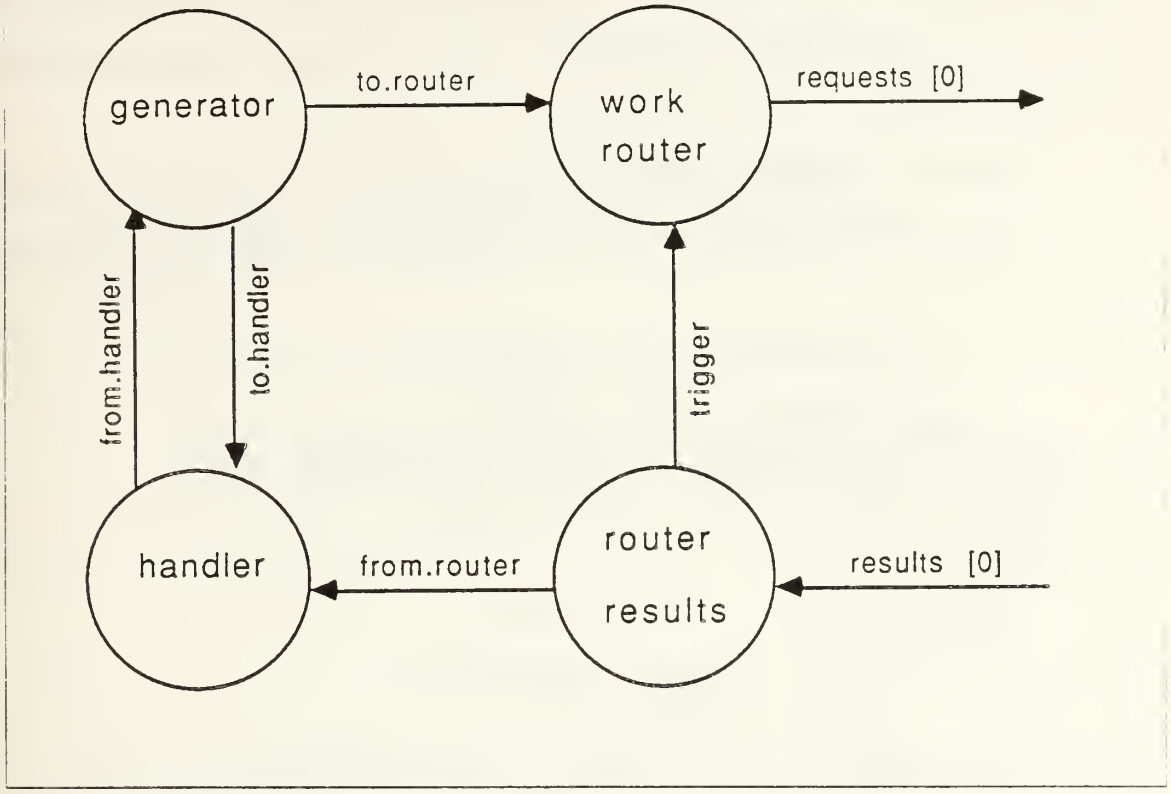


Figure 3.7

The Controller

```
CHAN OF ANY to.router, from.router:
CHAN OF ANY to.handler, from.handler:
CHAN OF ANY trigger:
PAR
  generator (to.router, to.handler, from.handler)
  work.router (to.router, trigger, requests)
  results.router (from.router, trigger, results)
  handler (from.router, to.handler, from.handler)
```

Figure 3.8

Controller Code

counter. The work router will only accept bundles from the generator while the counter does not exceed the farm bundle capacity. The code to implement this valve function is shown in Figure 3.9.

When the handler receives the last of the result bundles it signals the generator that the problem has been completed.

```
--- Work.router
VAL   farm.capacity IS (num.Transputers * 2) - 1:
BOOL  bundle.done:
INT   count:
SEQ
  count := 0
  WHILE TRUE
    PRI ALT
      trigger ? bundle.done
        count := count - 1
      (count <= farm.capacity) & to.router ? bundle
        SEQ
          requests ! bundle
          count := count + 1

--- Results.router
VAL  bundle.done IS TRUE:
WHILE TRUE
  SEQ
    results ? bundle
    PAR
      trigger ! bundle.done
      to.handler ! bundle
```

Figure 3.9

Valve Function Code

A single node on the farm consists of five processes: a work router, a result router, a work buffer, a result buffer, and a calculation process. A single node is shown in

Figures 3.10 and 3.11. Notice that the calculation process is given low priority while the four communication processes run at high priority, as explained in Chapter II.

```
CHAN OF ANY   to.buffer, to.calculation:
CHAN OF ANY   from.calculation, from.buffer:
CHAN OF BOOL  signal:
PRI PAR
  PAR
    work.router (requests.in, requests.out,
                 to.buffer,signal)
    work.buffer (to.buffer, to.calculation)
    result.buffer (from.calculation, from.buffer)
    result.router(from.buffer, results.in, results.out)
  calculation (to.calculation, from.calculation)
```

Figure 3.10

Single Node Code

The work router accepts bundles from the requests-in channel. If the work buffer is full, the work router relays the bundle to the next node in line by way of the requests-out channel. If the work buffer is empty, the bundle is sent there and the work buffer full flag is set.

The work and result buffers are present to decouple the communication in the work and result router processes from the calculation in the calculation process, as explained in Chapter II.

The work buffer holds a single bundle at a time. When the calculation process accepts a bundle from the work buffer, the work buffer signals the work router that the buffer is empty.

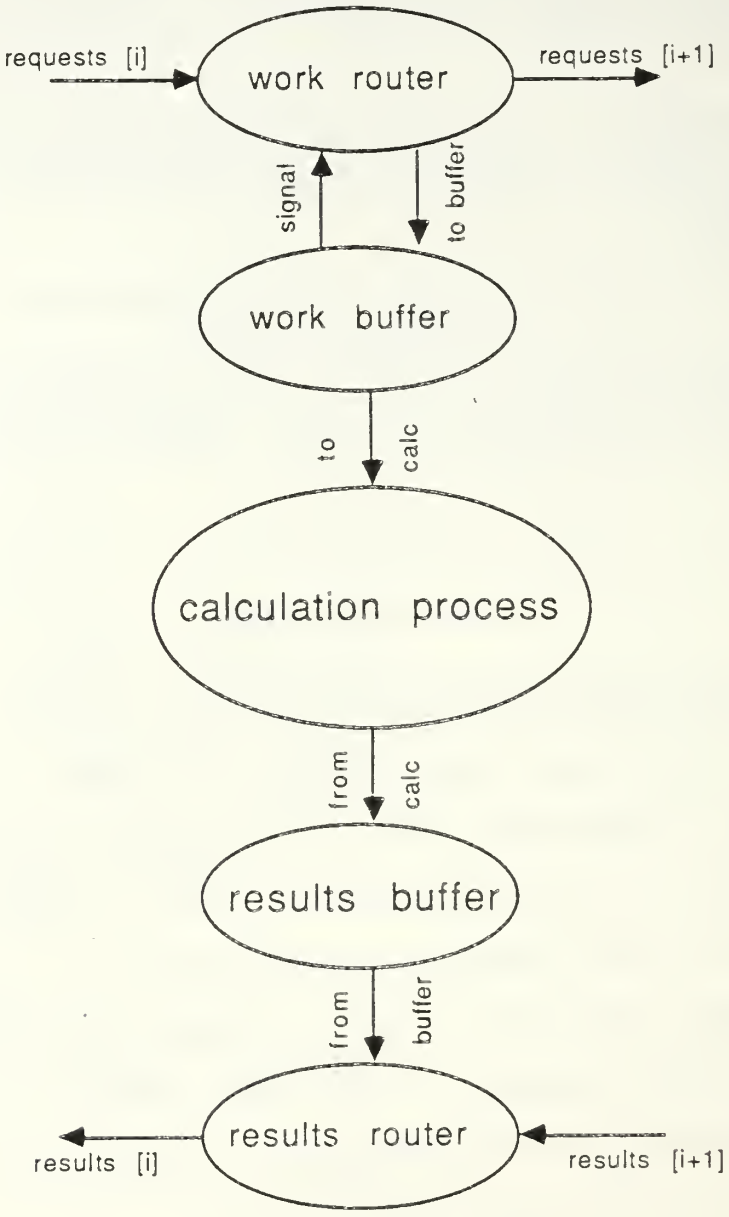


Figure 3.11

Single Node

The calculation process is where all work occurs. There, upon arrival, a bundle is separated into packets. The packets are processed sequentially and the results grouped into a result bundle. After the result bundle has been passed to the result buffer, the calculation process is ready to accept another bundle. Since the calculation process is sequential, it could be coded in a programming language other than OCCAM such as Ada, Pascal, or C and inserted into the OCCAM workfarm harness.

The results buffer is a pure buffer that relays a result bundle to the result handler and waits for another result bundle to arrive.

The results router receives result bundles either from the result buffer or the results-in external channel. In either case, the bundle is discharged along the results-out channel. An arriving bundle from the buffer is given priority over the external channel so that the calculation process will be free for more work as soon as possible.

There has to be some device to allow for matters of initialization and reporting in the farm. The method used in this thesis was to place a tag at the head of every bundle array. Depending on the tag, a bundle could be of the following types: setup, data, or report. Generally, upon arrival of a bundle, the tag is examined, and action is taken on the bundle accordingly.

## D. TOPOLOGY

A workfarm is not limited to the linear array; trees are an attractive option. Because of communication path lengths, trees would seem better suited for a workfarm with large numbers of Transputers. Consider linear array and binary tree workfarms, each with 100 farm nodes, for example. Assuming the problem was computation intensive enough so that all 100 nodes would be utilized, a bundle would have to travel through 99 nodes to get to the 100th node in the linear array. In the binary tree, however, a bundle would have to travel through at most six nodes to reach any of the 100 nodes. A trinary tree would lower the communication overhead further. Given a workfarm of the same number of Transputers working the same problem, trees are more efficient than linear array in terms of utilizing each Transputer.

Trees were not used in the research for a number of reasons, however. First of all, there were not enough Transputers available to conduct research and achieve significant results. A binary tree workfarm was implemented but the performance gains were so miniscule that the author believes that significant gains would not be realized until large numbers of Transputers, perhaps 50 or more, were used. The modification to the linear array workfarm algorithm to achieve binary tree topology was slight and only occurred in the work and result routing processes of the node process.

The actual code is listed in Appendix A. Secondly, the B003 boards, with four Transputers each, are not well suited to implementing trees since half of the 16 links are hardwired.

A large number of processors must be arranged in a regular fashion or the configuration code and communication algorithms become too complex. Commonly used regular structures are arrays (1D, 2D, 3D, ...), trees (binary, trinary, etc.), and hypercubes. More complex but still regular structures have been proposed such as hypernet [HwGh87]. Such structures appear promising for a networks of large numbers of Transputers.

# IV. WORKFARM PERFORMANCE

## A. GENERAL

The performance of a workfarm is dependent on the following factors: the number and speed of the Transputers, the number of packets in the problem, the number of computations per packet, the packet size, and the number of packets per bundle. To determine the relationships between these factors and ultimately the number of Transputers required to complete a problem in a certain amount of time, research was conducted on a standard workfarm using a variable problem.

## B. ENVIRONMENT

The research in this chapter was conducted on a workfarm with a physical configuration as pictured in Figure 4.1. The number of Transputers utilized in the farm portion was varied from one to eight. The speed of an individual Transputer depended on its respective type. The controller process was placed on a remote T414-15 Transputer. The T414 on the B004 board was used only for the compiling, extraction, and loading of code to the workfarm network. The farm nodes were placed on T800 Transputers.

A B002 board with its T414-15 and RS232 interface was utilized so that results could be displayed to a VT220 terminal. The B007 board was included in the network for any

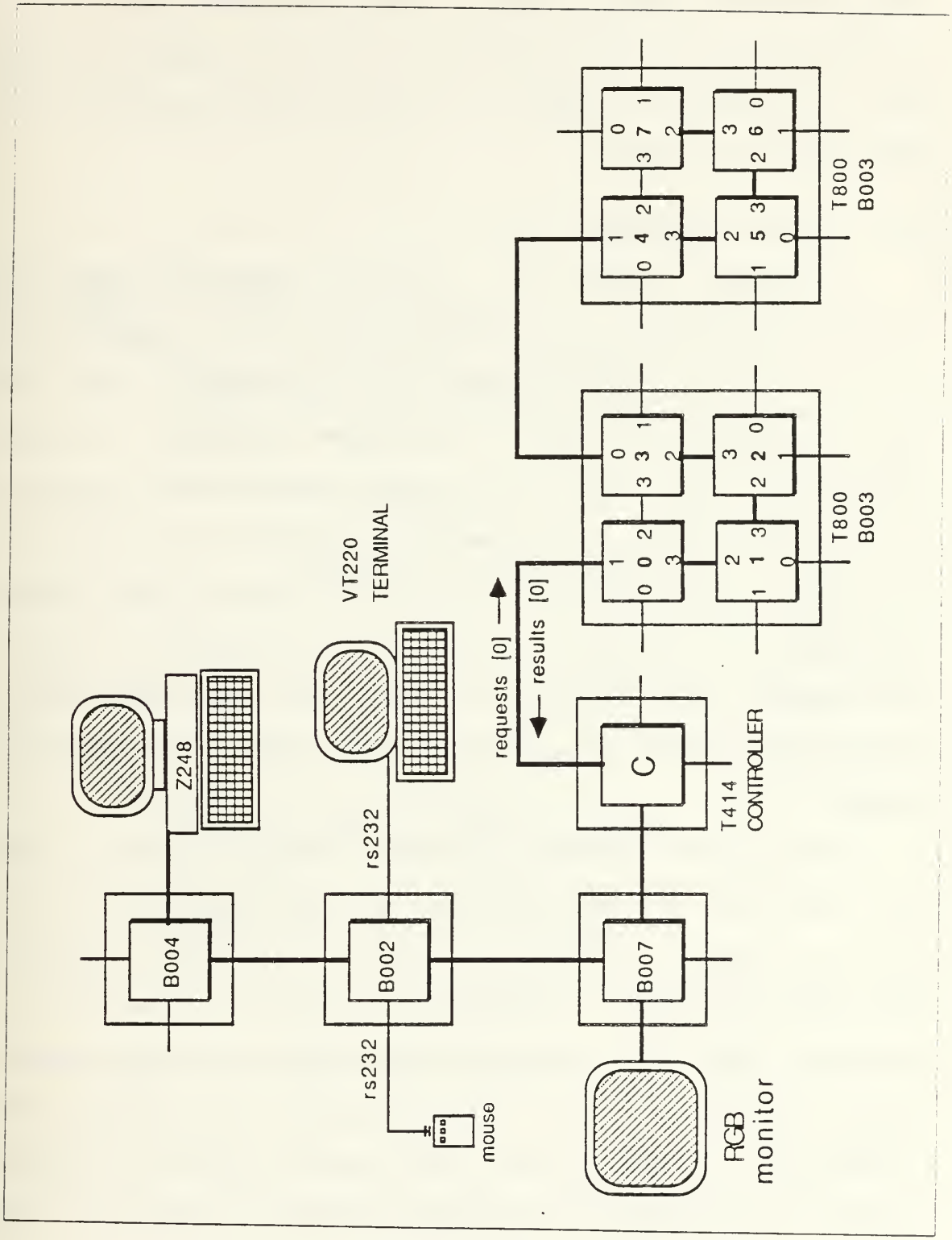


Figure 4.1

Workfarm Physical Configuration

future graphic work. It acted solely as a relay and did not figure in any of the testing. The entire workfarm network ran at a common link speed of 20 Mbits/second except for the B004 and B002 boards.

## C. TESTING APPARATUS

To change problems on a workfarm requires modification of the calculation process of the farm nodes, the generator process, the handler process, and the message arrays that pass throughout the farm. To test the workfarm, it was necessary to be able to vary the problem size quickly and easily. The problem consisted of a variable number of 32 bit floating point (REAL32) multiplications and the constant communication overhead of the workfarm algorithm. When the farm nodes were initialized, each farm node was passed an integer representing a certain number of calculations per packet.

The generator process produced a certain number of packets which were grouped into bundles and distributed to the farm, as shown in Figure 4.2. When a calculation process in the farm received a bundle, it simulated separating the packets from the bundle and did "calculation.per.packets" REAL32 multiplications. The calculation process simulated assembling a result bundle and was then ready for the next bundle. This testing calculation process is shown in Figure 4.3.

```
tag := data
SEQ i = 0 for total.bundles
  SEQ
    SEQ j = 1 FOR packets.per.bundle
      [bundle FROM (j TIMES 4) FOR 4] :=
      [dummy.array FROM 0 FOR 4]
    requests ! bundle
```

Figure 4.2

Generator Process Code

```
SEQ i = 1 FOR packets.per.bundle
  SEQ
    [dummy.array FROM 0 FOR 4] :=
    [bundle.in FROM (i TIMES 4) FOR 4]
    SEQ j = 0 FOR calcs.per.packet
      x := x*x
    SEQ j = 0 FOR 4
      bundle.out [j] := dummy(BYTE)
```

Figure 4.3

Calculation Process Code

A timer was started preceding the initialization phase. When the handler process received the last result bundle the problem was completed and the timer was stopped. The time required to complete the entire problem was simply the stop time minus the start time.

During the report phase, the handler received packets from the farm nodes that contained the number of bundles that each node processed and the node identification. The handler passed this information and the problem completion time to the VT220 terminal for display.

The bundle used for communication in the workfarm consisted of an array of bytes. The first four bytes were retyped to represent the integer tag. The next eight bytes were retyped into a setup array of two integers to hold initialization values. When the bundles were used in their normal role of carrying packets, they were essentially "dummy" arrays, as the only meaningful information in each bundle was the tag. Figure 4.4 lists the bundle declaration.

```
VAL packets.per.bundle IS 10000/total.bundles:
VAL bundle.size        IS packets.per.bundle + 1:
VAL packet.size.int    IS 1:
VAL bundle.size.byte   IS bundle.size.int TIMES 4:
VAL packet.size.byte   IS packet.size.int TIMES 4:

[bundle.size]BYTE      bundle:
INT    tag             RETYPES [bundle FROM 0 FOR 4]:
[]INT  setup.array     RETYPES [bundle FROM 4 FOR 8]:
```

Figure 4.4

Bundle Declaration

The packet size used for the research was one integer. This represented a unit packet with four byte-size parameters. Most problems will have packets consisting of some multiple of four bytes; for example, the Mandelbrot Set problem has 12 byte request packets and 32 byte result packets.

## D. RESULTS AND CONCLUSIONS

Figure 4.5 illustrates some of the relationships that exist in an eight Transputer workfarm among the many variables that affect performance. The vertical axis shows the time to calculate 10,000 packets. The horizontal axis indicates the number of REAL32 multiplications each packet represents. If calculations.per.packet is 100, the total workload is 1,000,000 REAL32 multiplications.

Each line of the graph represents a different bundle size, in terms of packets per bundle. For example, a bundle made up of 16 packets is 17 integers long (68 bytes): 16 one integer packets and a one integer tag.

Each line begins flat and at some point begins increasing linearly. The flat line indicates two things. While the line is flat, not all eight Transputers are being utilized. For small bundles of one packet per bundle, the farm is underutilized until the workload reaches 120 calculations per packet. As the bundle size increases, more Transputers are utilized sooner. For example, at 16 packets per bundle,

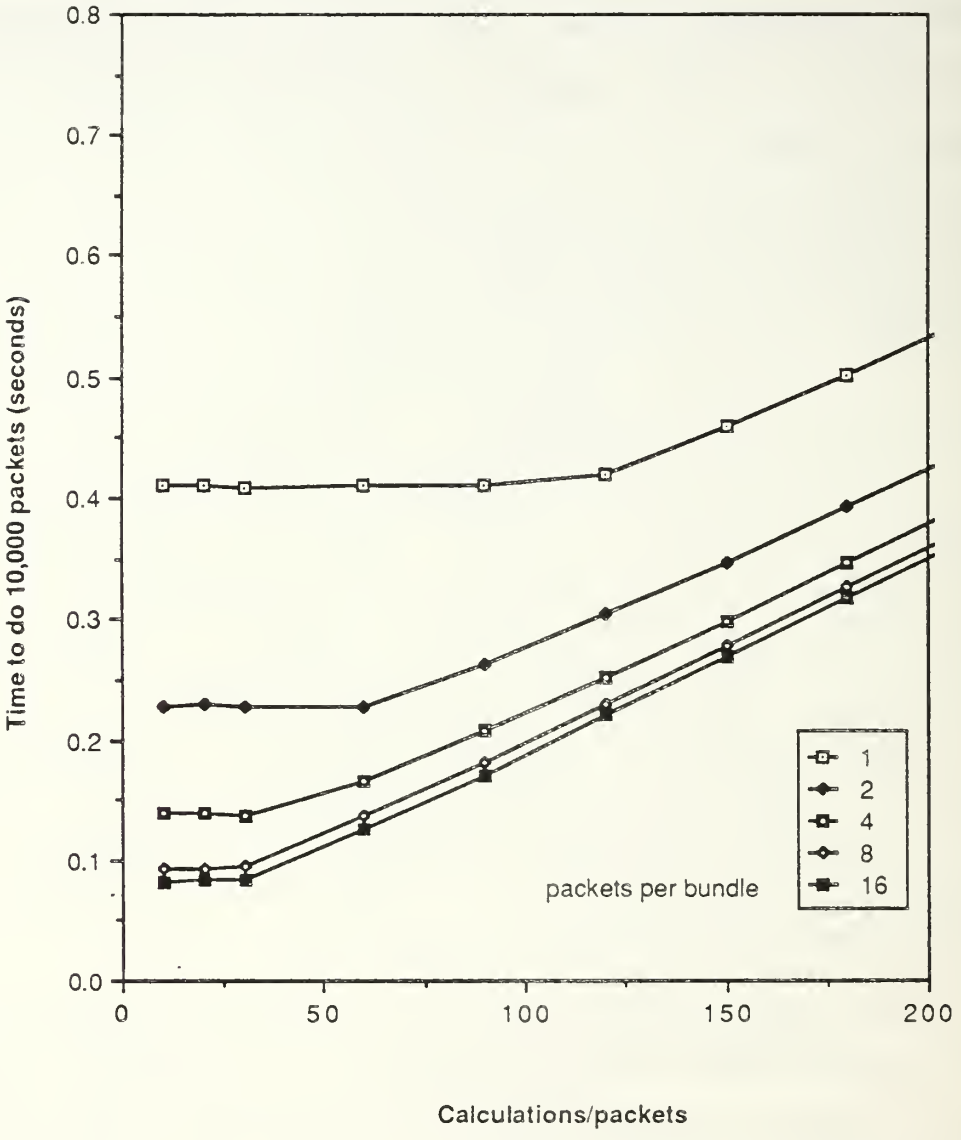


Figure 4.5

Workfarm Performance Characteristics

the workfarm is fully utilized at approximately 30 calculations per packet.

The flat line also means that the workfarm is completing an increasing number of calculations with no corresponding increase in time. This is explained by observing how many Transputers were utilized at each calculation point. As the workload increased, the workfarm utilized more of its available processors. Of course, there came a time when all available Transputers, in this case eight, were in use. From then on, the time required increased proportionally to the increase in the workload.

Once all the farm Transputers were in use, and the workload continued increasing, the time increased at a constant rate. This rate was equal for all bundle sizes.

The question of time to complete a problem can best be represented by the following equation:

```
time := (calcs.per.packet * total.packets) + comm.ovhd
```

With a small number of calculations per packets, the communication overhead is a significant factor in the equation. As the workload increases, the communication overhead decreases in significance.

The time required to complete the problem decreased as the bundle size grew. At greater than 16 packets per bundle, however, the time reductions became negligible. This was not surprising, considering that with one packet per bundle,

half of the bundle was used for communication overhead (one integer for the tag, one integer for the packet). At 16 packets per bundle, however, 1/17 of the bundle was overhead, or 5.9 percent. Between one packet per bundle and 16 packets per bundle there was a large difference in the overhead percentage; the reductions in time was correspondingly great. The overhead percentage difference between 16 and 32 packets per bundle was small, however, and the reduction in time required was also correspondingly small.

In each farm node process, one bundle is declared in each of the four communication processes and two bundles are declared (bundle.in and bundle.out) in the calculation process. Sixteen packets per bundle (17 integers) is a good bundle size since the bundle is large enough to yield near optimal performance yet small enough to require little on-chip memory. Bundle sizes greater than 667 bytes will leave no room on the T800 on-chip memory for any other data structures because six bundle arrays will require 4002 bytes of memory which is close to the T800 on-chip memory capacity of 4096.

With 100 calculations per packet and 10,000 packets, the total workload is 1,000,000 REAL32 multiplications plus the communication overhead. Dividing the workload by the total time to complete the workload yields "workfarm mega-floating point operations per second" (WF-MFLOPS). Plotting WF-MFLOPS versus number of Transputers yields the graph in Figure 4.6.

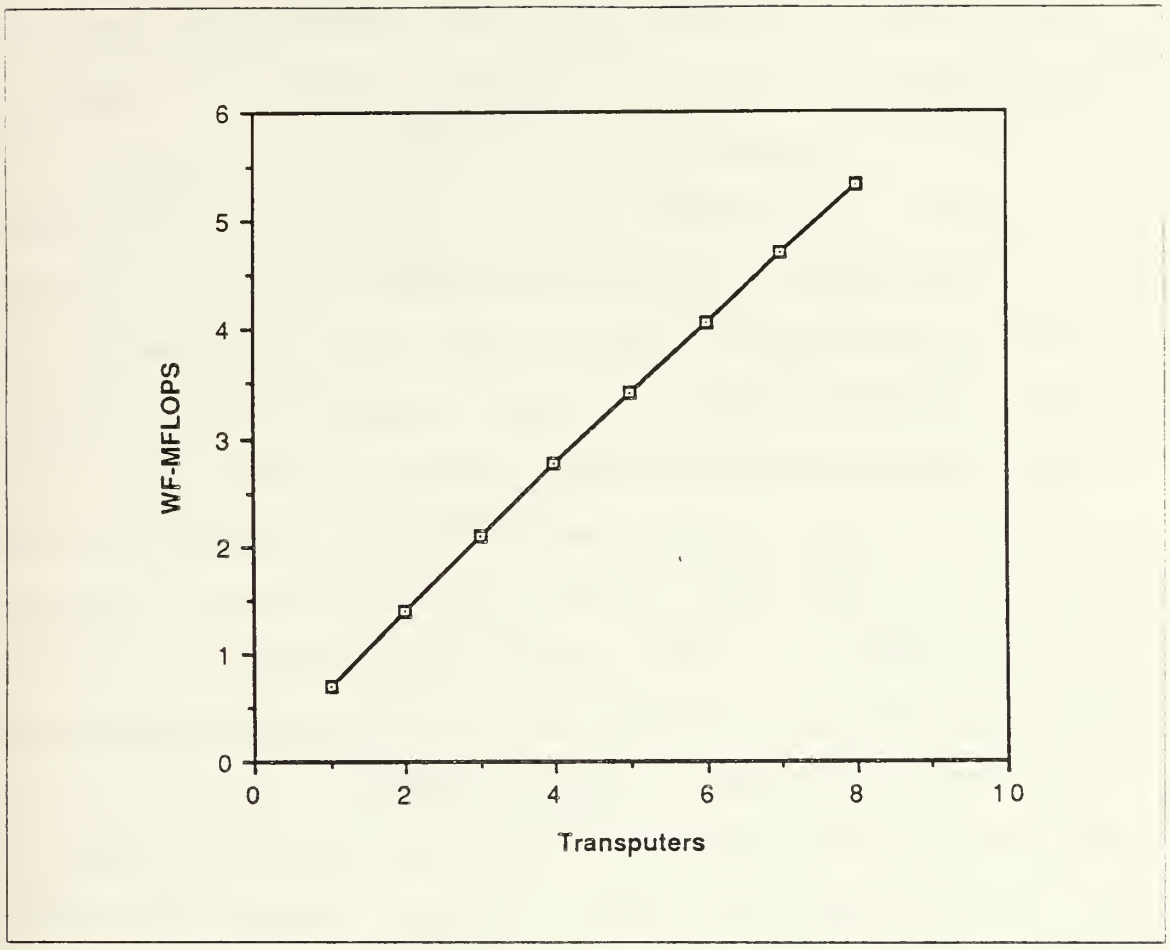


Figure 4.6

Linear Performance

The resulting nearly straight line indicates that the work-farm is achieving near linear performance; that is to say, WF-MFLOPS is directly proportional to the number of Transputers in the farm.

It should be realized that at some number of Transputers, the line in Figure 4.6 will turn sharply horizontal. Where on the graph this occurs depends on the workload and number of Transputers. It represents the point at which the controller cannot provide request bundles fast enough to

keep all the farm nodes busy. Adding more Transputers to the farm beyond this point is wasteful, since no further increase in work capacity can ever occur.

It would be helpful to be able to predict either how many Transputers in a farm are needed to solve a problem in a certain amount of time or how long a problem will take given a certain number of farm Transputers. To do so, one has to realize that a workfarm can be in one of two limiting conditions, depending on the workload and number of Transputers. The first case is when the workfarm is "calculation limited"; that is, the ultimate performance is limited by the workload, not by the controller request bundle generation rate. The second case is when the workfarm is "communication limited"; with small workloads, the farm nodes are able to complete request bundles faster than the controller can supply them.

The workfarm performance can be characterized by a set of equations using the notation in Table 4.1. Figure 4.7 is provided as a reference.

The time to solve a problem on a workfarm (T) is obviously

$$T = \frac{N}{r}.$$

And, since $r \leq m$,

$$T = \frac{N}{r} \geq \frac{N}{m}.$$

TABLE 4.1

WORKFARM VARIABLES

| Variable | Represents |
|---|---|
| B | single node maximum calculation rate - bundles/second with no degradation |
| r | controller limiting rate - bundles/second |
| m | controller maximum rate - bundles/second |
| $a_i$ | bundles/second accepted by node i for calculation |
| T | time to solve complete problem |
| N | total number of bundles in problem |
| n | total number of nodes in farm |
| $D_x$ | degradation of B caused by x bundles/second |

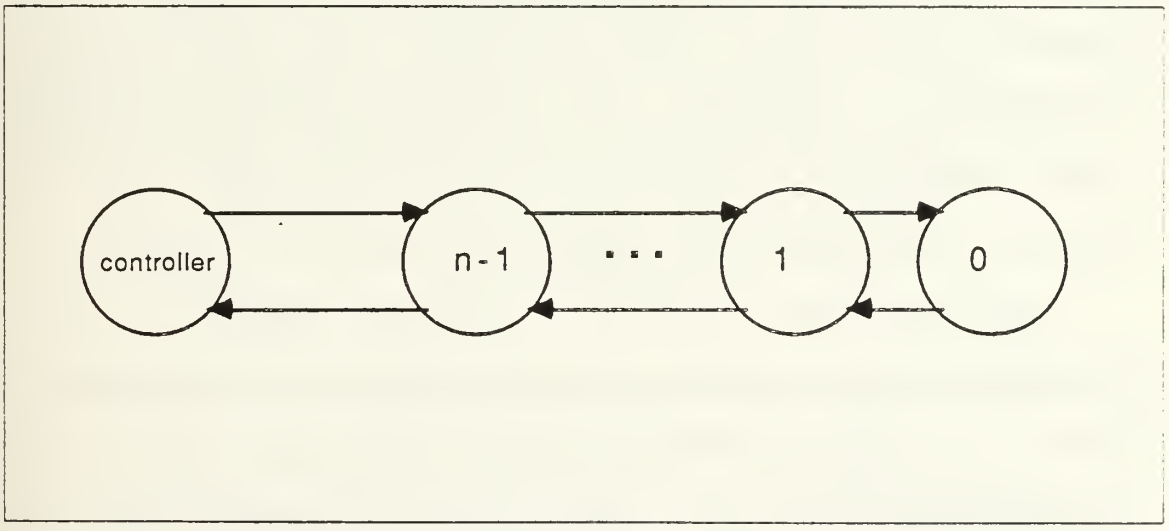


Figure 4.7

Workfarm Reference

With n farm nodes, each node accepts the following number of bundles per second for calculation:

$$a_0 = B$$

$$a_1 = B \cdot D_{a_0}$$

$$a_2 = B \cdot D_{a_0+a_1}$$

$$\vdots$$

$$a_{n-1} = B \cdot D_{a_0+a_1+\ldots+a_{n-2}}.$$

Thus,

$$m \geq r = \sum_{i=0}^{n-1} a_i.$$

The total number of bundles (N) in a problem is a known factor that can be used for predictions. The rate that bundles can be accepted and processed by a single Transputer node without degradation (B) can also be determined beforehand by measuring the maximum number of bundles that a single farm Transputer can process in a certain time period.

Consider the calculation limited workfarm, which is characterized by a large workload. There are two predictions that can be made. First of all, if there is a certain number of Transputers available, one can determine how long it will take to complete the problem. Secondly, if one has a time limit in which to complete the problem, one can determine

how many Transputers will be required, assuming the problem does not become controller limited.

Research showed that when the farm was calculation limited, each farm node processed approximately the same number of bundles, with a slight increase in processed bundles from first to last node. This corresponds to the decreasing degradation in the farm as the nodes further from the controller have to route less and less bundles. If the number of Transputers (n) is known, and relatively few Transputers are used, the number of bundles processed by each farm node is approximately N ÷ n or $N_i$. Since $a_i$ is approximately the same for each node (discounting degradation) and the farm is calculation intensive, the overall time required by one node to complete $N_i$ bundles can be assumed to be approximately the same as the overall time taken to complete the entire problem. This simple method to predict the overall time of a computation limited workfarm was effective to within five percent of the actual results. The resulting equation is:

$$T = \frac{N}{n \cdot B}.$$

To complete the problem in a given amount of time, one can calculate approximately how many Transputers will be required by merely switching T and n in the above equation:

$$n = \frac{N}{T \cdot B}.$$

The communication limited case is characterized by a low workload with the farm nodes accepting request bundles at a greater rate than in the calculation limited case. Obviously, the more request bundles accepted, the more result bundles generated; and in general, a greater percentage of processor time is spent passing bundles. The lower the workload, the sooner the farm will be able to exceed the controller's ability to supply bundles. If the first few farm nodes can match the rate at which the request bundles come from the controller, then nodes further down the line in the farm simply will not receive any work. Thus, the controller generation rate becomes the limiting factor as r decreases from m. This generation rate includes the overhead caused by the controller having to accept incoming result bundles. The number of nodes in the farm that do useful work is dependent on the capacity of the controller to supply request bundles.

It would be useful to know at what point a particular problem becomes communication limited; that is, for any problem, what is the maximum number of nodes in the farm that will do useful work. The workload and total number of bundles are known from the problem description. A rough estimation may be reached by assuming that at equilibrium, where some number of farm nodes are able to match the controller's request bundle generation rate (r), each working node is processing approximately the same number of bundles. Dividing r by the single node maximum calculation rate (B)

yields the approximate number of nodes that will do useful work. Unfortunately, r is difficult to ascertain without implementation and testing. An upper bound may be obtained by substituting m for r. By testing on a workfarm with a single farm node, as shown in Figure 4.8, m can be determined. The single workfarm node merely accepts request bundles and returns to the controller a simulated result bundle without doing any calculations. Thus an upper bound for the controller capacity is easily measured.

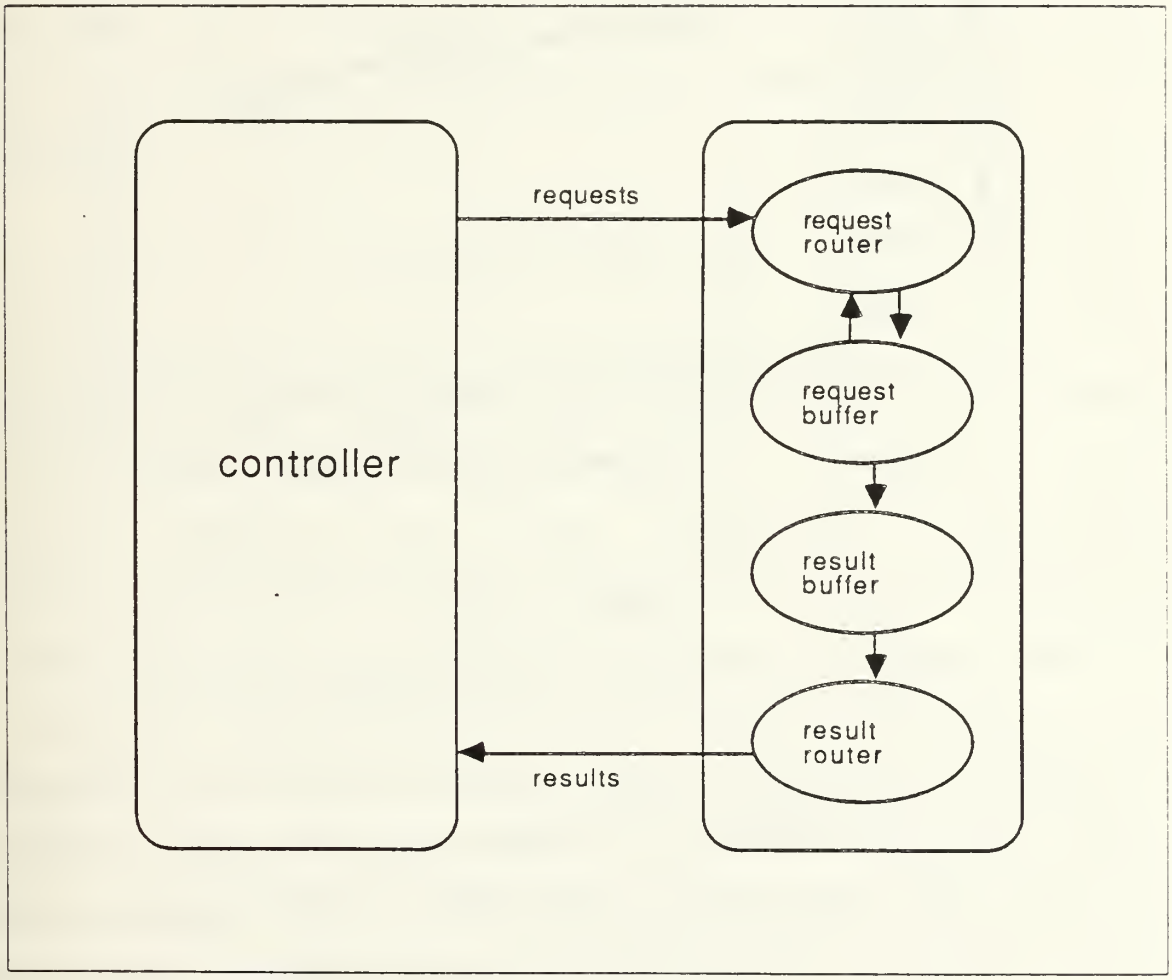


Figure 4.8

Single Farm Node Test

Figure 4.9 shows the actual controller request bundle generation rate (r) versus increasing workload for a farm consisting of 15 T414 nodes.

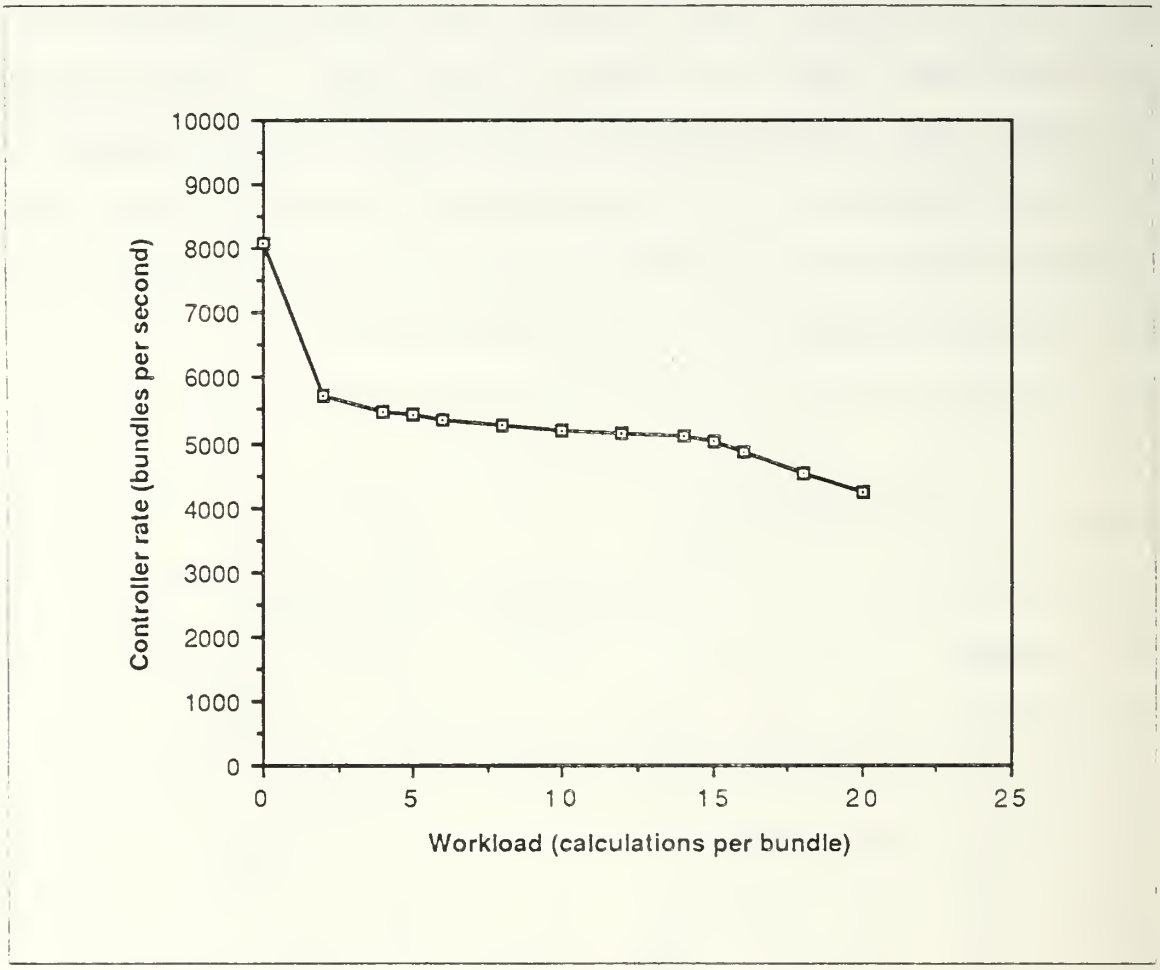


Figure 4.9

Controller Request Bundle Generation Rate

The initial rate versus a workload of zero is a high 8000 bundles per second. This zero workload rate, converted to 550 Kbytes per second, approaches the theoretical maximum unidirectional link rate of a T414, 750 Kbytes per second, and is in fact m. The zero workload rate will never match

the theoretical maximum rate because of the controller overhead. When a load is put on the farm, the actual rate drops off sharply to approximately 5700 bundles per second and then decreases gently at a nearly constant rate of approximately 100 rate units per workload unit. This rate holds until the farm begins to become calculation limited and the rate drops sharply again.

Since r is relatively constant during the communication limited portion of the graph, it can be used in the following rule of thumb equation for determining the maximum number of useful farm Transputers in a communication limited workfarm:

$$\text{Transputer limit} = \frac{r}{B}.$$

Again, m can be substituted for r in this equation to determine an upper bound on the number of Transputers a particular workload will utilize. Determining r accurately without testing the actual problem on a farm of multiple Transputers was a difficult problem. The only success in obtaining an approximate value of r was to test the actual problem on a farm of at least four Transputers. The resulting r value could then be used to project a Transputer limit for problems with larger workloads. The validity of this projection is a function of the accuracy of r and B.

Any time the number of Transputers in the farm is less than r ÷ B, those Transputers are going to be fully utilized, each processing approximately N ÷ n bundles.

# V. PREDICTIONS

## A. GENERAL

The research conducted in Chapter IV can be used to predict the performance of the workfarm in some cases. Two different problems are used in this chapter as examples: coordinate transformation and Mandelbrot set drawing. Actual performance results from these two problems were compared with the predictions.

## B. COORDINATE TRANSFORMATION PROBLEM

This coordinate transformation problem originated from research for a autonomous walking machine [Ri87]. The following is a brief description of the problem and its implementation on a workfarm. For in-depth coverage of the problem itself, the reader is referred to [Ri87].

An autonomous land vehicle, known as the Adaptive Suspension Vehicle, possesses an optical radar scanner which the vehicle uses to "see" the forward terrain. The scanner returns range measurements for each elevation and azimuth increment in its scan. A single scan consists of 128 azimuth increments for each of 128 elevation increments, a total of 16,384 iterations. The azimuth, elevation, and range are combined with six other inputs from the vehicle's inertial navigation system to develop a cartesian coordinate position of the particular scanned point. The elevation of every

point in the scan is kept in a terrain matrix data structure.

For the workfarm implementation, each packet represented one scan iteration and consisted of three bytes representing the scanner azimuth angle, the scanner elevation angle, and the resulting range. There were 16384 packets in one scan.

The vehicle's inertial navigation system (INS) supplied the vehicle attitude and position information, for a total of six inputs. The attitude information consisted of the vehicle's azimuth angle, pitch angle, and roll angle. The positional information consisted of the vehicle's x, y, and z transformation (distance) from the INS reference point. Because these six inputs remained constant for the entire scan, they could be passed to the farm and processed as much as possible during the initialization phase of the workfarm.

The radar scanner was easily implemented on a separate "scanner" process. Byte range values for a flat, zero elevation scan were calculated and then sequentially passed to the controller process through a channel. Although not implemented, the rate of outgoing range values could be easily controlled through use of the Transputer timer.

Each of the 16384 packets in a single scan were processed in the following way. Each byte in the packet was first hashed into a useful 32-bit floating point real number (REAL32). For example, the scanner azimuth range was -40 to +40 degrees. Since a byte can only represent the numbers 0

to 255, the scanner azimuth byte had to be converted into the correct data upon arrival in a farm node. Each bundle contained a byte tag and 128 packets.

For each packet the three new parameters were combined with the six parameters received during the initialization phase using a Denavit-Hartenberg (D-H) transformation to yield the x, y, and z coordinates of the particular spot being scanned [Ri87]. These three coordinates were themselves converted into bytes, loaded into the result bundle, and eventually passed back to the controller.

The problem was implemented on a standard workfarm as described in Chapters III and IV; that is, the controller process was placed on a T414-15 and eight farm nodes were placed on eight T800s. The scanner process was placed on a separate T414-15 and shared a single link with the controller process. The calculation process of a farm node is listed in Appendix C.

Testing on a single T800 Transputer yielded an approximate calculation rate (B) of 43.63 bundles per second. Total bundles for one scan (N) was 128.

The actual time for the workfarm to process a single scan (T) was 0.427 seconds, including loading the resultant 16384 altitude values into a terrain map matrix. The farm was controller limited, with 6.3 of the 8 Transputers being utilized.

The actual controller request bundle flow rate (r) was 300 bundles per second. If this value of r could have been estimated correctly beforehand, the predicted Transputer limit for this problem would have been r divided by B or 6.9 Transputers. This is very close to the 6.3 Transputers that were actually utilized.

## C. MANDELBROT SET

The drawing of the Mandelbrot Set is a problem that demonstrates the best qualities of the workfarm. It is a problem that is extremely computation intensive and where the amount of work each packet will entail is unknown.

For an in depth description of the Mandelbrot Set problem, the reader is directed to [Po86]; this chapter mainly discusses implementation and performance.

> Essentially, the Mandelbrot set is generated by iterating a simple function on the points of the complex plane. The points that produce a cycle (the same value over and over again) fall in the set, whereas the points that diverge (give ever-growing values) lie outside it. When plotted on a computer screen in many colors (different colors for different rates of divergence), the points outside the set can produce pictures of great beauty. [Po86]

The problem is divided into independent work packets; each packet containing an integer tag and two other integers that represent a coordinate position on the complex plane. As stated before, the packet workload is variable; there is no way of knowing beforehand how much calculation each packet will require. Each packet really represents 16 coordinates because the farm node uses the single coordinate in

the packet as the starting point for 15 more consecutive horizontal coordinates. Each iteration can entail from one to 256 loops of approximately ten arithmetic operations each.

A factor in this particular problem is that the handler in the controller has to pass result bundles to the graphics routine on the B007 graphics board where the results are drawn. The controller flow rate (r) is lowered because of this overhead.

The implementation of the Mandelbrot set onto the workfarm is somewhat different than in previous implementations. There is no benefit in bundling together packets to improve communication efficiency because of the variable packet workload. A request packet is already fairly large, 12 bytes, and a result packet is even larger, 32 bytes. Both request and result packets represent 16 coordinate points which can represent a massive amount of computation.

The generator and handler processes of the controller and the calculation process of the farm node are listed in Appendix D.

The problem was implemented on the same workfarm configuration as the coordinate transformation problem with the exception of the scanner.

The coordinate matrix was 512 by 512; thus there were 16384 packets (N), each representing 16 horizontal coordinates as stated before.

When each coordinate only required one loop iteration, a solid dark gray screen was drawn in 1.5 seconds. The farm was controller limited as only 3.2 of the 8 Transputers were utilized. When each coordinate required 256 iterations, the workfarm required 81.5 seconds to draw a solid black screen. The farm, of course, was computation limited and all eight Transputers were utilized with only a slight variation in the number of packets processed by each.

Because the packet workload is variable, predictions are possible only when each coordinate represents the same amount of loop iterations; i.e., the screen is solid black (256 iterations per coordinate) or solid dark gray (one iteration per coordinate).

Testing on a single T800 Transputer yielded an approximate calculation rate (B) of 39 packets per second. Using this value in the calculation limited equation to predict the actual time to draw a solid black screen on a farm of eight T800 Transputers yielded 52.5 seconds. As noted previously, however, the actual time was 81.5 seconds. This discrepancy arose because the controller was limiting the farm although the problem was still calculation limited. The controller limited the farm because the controller's handler had to send every results packet to the graphics Transputer for drawing. This caused the controller to wait because the graphics process would not accept another packet

until the previous one had been completed (drawn). This waiting translated into significant overhead.

The actual problem was again tested, this time without the controller handler having to pass results to the graphics Transputer (no picture was drawn); the controller handler merely accepted results from the farm. The actual time in this case was approximately 53 seconds, very close to the original prediction.

Clearly, the equations do not work if the controller has to do work on the results after reception. In this case, for the equations to remain applicable, the controller handler needs to relay work to a buffer Transputer between the controller and the graphics Transputer (for example). Or, if much work of a different type is needed, the results could be passed to another, separate farm.

# VI. CONCLUSIONS AND RECOMMENDATIONS

## A. CONCLUSIONS

This thesis deals primarily with the work distribution algorithm known as workfarm. Although limited to problems divisible into independent work packets, it is a simple yet extremely effective way of processing in parallel and achieving near linear performance speedups. Processing rapid streams of data from a weapon system sensor would seem to fall into the workfarm category of problems. The coordinate transformation example demonstrates that the workfarm is well suited for a radar problem.

## B. RECOMMENDATIONS

Although the equations for workfarm performance have been developed in this thesis, research on how to accurately estimate the controller request bundle flow rate for a farm of a given number of Transputers and given workload needs to be done to predict the limit at which a farm becomes controller limited. For the same reason, research is needed to accurately estimate the degradation factor for each node in the farm.

Both the pipeline and workfarm are good for specific types of problems; however, much work remains in developing and evaluating new work distribution algorithms for other

kinds of problems. Specifically, [HwCh87] appears promising as a source for new work distribution algorithms.

Much work remains, too, in development and evaluation of new network physical topologies. For example, in a workfarm, a simple binary tree farm topology would allow a controller to supply the farm with a far greater rate of request bundles than it could to a linear array farm. Other topologies, such as hypernet [HwGh87], would be extremely interesting to implement. Currently there are too few Transputers in the lab to significantly explore these different topologies; perhaps, more topology research can be done when greater numbers of Transputers are available. The relative ease to configure multi-Transputer networks by virtue of the OCCAM programming language makes widespread research in network topology practical for the first time.

An ADA compiler will be available for the Transputer family soon; since ADA is the Department of Defense standard programming language, research on the implementation of Ada on Transputers should begin as soon as the compiler arrives.

The ultimate goal of an ongoing series of Transputer theses at the U.S. Naval Postgraduate School, of which this thesis is part, is to develop an alternative computer architecture for the U.S. Navy Aegis Combat System. The expertise base represented by this series of theses has reached the point where the next step should be the simulation of the AN/SPY-1A 3D Phased Array Radar Controller, the main

component of the Aegis Combat System. It seems likely that the workfarm would have some utility in such a simulation.

One fact stands out from working with Transputers: the Transputer system is revolutionary. Its performance jump over anything short of a supercomputer is orders of magnitude. True parallel processing is implemented easily with a high level programming language, employing the best elements of software engineering. The Transputer system seems especially useful for the military, considering the Transputers suitability for embedded control applications and parallel processing networks.

APPENDIX A

## BINARY TREE WORKFARM SOURCE CODE

The only difference between the binary tree and the linear array workfarms occurs in the request and result routers of the farm node. To keep the program compact, two separate versions of the farm node were developed, a fork node and a leaf node. That is all that is listed in this appendix. The complete algorithm for a linear array workfarm is in Appendix B.

```
--------------------------------------------------------------
PROC fork(CHAN  OF ANY    requests.in, requests.out.left,
                          requests.out.right,
                          results.in.left, results.in.right,
                          results.out,
          VAL INT         proc.id)
--------------------------------------------------------------
  CHAN OF ANY  from.result.router, signal:
  CHAN OF ANY  to.buffer, to.calculation:
  CHAN OF ANY  from.buffer, from.calculation:

  PRI PAR
    PAR
      VAL left IS FALSE:
      VAL right IS TRUE:
      ... communication
    ... calculation
:
```

```
--------------------------------------------------------------
                    fork request router
--------------------------------------------------------------
--- declarations
[bundle.size]BYTE bundle:
INT    tag RETYPES   [bundle FROM 0 FOR 4]:
BOOL   d.buffer.empty, switch:
INT    num.left, num.right:

SEQ
  --- initialization
  d.buffer.empty := TRUE
  IF
    proc.id = 0
      SEQ
        num.left  := 6
        num.right := 6
    proc.id > 0
      SEQ
        num.left  := 2
        num.right := 2

  WHILE TRUE
    PRI ALT
      ALT
        signal ? d.buffer.empty
          SKIP
        from.result.router ? switch
          IF
            switch = left
              num.left := num.left + 1
            switch = right
              num.right := num.right + 1
      requests.in ? bundle
        IF
          tag = data
            IF
              d.buffer.empty
                SEQ
                  d.buffer.empty := FALSE
                  to.buffer ! bundle
              else
                IF
                  num.left >= num.right
                    SEQ
                      num.left := num.left - 1
                      requests.out.left ! bundle
                  else
                    SEQ
                      num.right := num.right - 1
                      requests.out.right ! bundle
```

```
          tag = setup
            PAR
              requests.out.left ! bundle
              requests.out.right ! bundle
              to.buffer ! bundle

          tag = report
            PAR
              requests.out.left ! bundle
              requests.out.right ! bundle
              to.buffer ! bundle
```

---

fork result router

---

```
[bundle.size]BYTE  bundle:
WHILE TRUE
  PRI ALT
    from.buffer ? bundle
      results.out ! bundle
    results.in.left ? bundle
      PAR
        from.result.router ! left
        results.out ! bundle
    results.in.right ? bundle
      PAR
        from.result.router ! right
        results.out ! bundle
```

```
-------------------------------------------------------------
PROC leaf(CHAN OF ANY    requests.in, results.out,
          VAL INT        proc.id)
-------------------------------------------------------------
  CHAN OF ANY  signal:
  CHAN OF ANY  to.buffer, to.calculation:
  CHAN OF ANY  from.buffer, from.calculation:
  PRI PAR
    PAR
      ... communication
    ... calculation
:

-------------------------------------------------------------
                      leaf request router
-------------------------------------------------------------
--- declarations
[bundle.size]BYTE bundle:
INT     tag RETYPES   [bundle FROM 0 FOR 4]:
BOOL    d.buffer.empty:

SEQ
  d.buffer.empty := TRUE
  WHILE TRUE
    PRI ALT
      signal ? d.buffer.empty
        SKIP
      requests.in ? bundle
        IF
          tag = data
            IF
              d.buffer.empty
                SEQ
                  d.buffer.empty := FALSE
                  to.buffer ! bundle
              else
                SKIP

          tag = setup
            to.buffer ! bundle

          tag = report
            to.buffer ! bundle

-------------------------------------------------------------
                      leaf result router
-------------------------------------------------------------
[bundle.size]BYTE  bundle:
WHILE TRUE
  SEQ
    from.buffer ? bundle
    results.out ! bundle
```

APPENDIX B

GENERIC WORKFARM SOURCE CODE

```
---  global variable file
VAL  MAXnumT8             IS    8:
VAL  total.bundles        IS    625:
VAL  total.packets        IS    10000:

VAL  packets.per.bundle IS    total.packets/total.bundles:
VAL  bundle.size.int      IS    packets.per.bundle + 1:

VAL  packet.size.int      IS    1:
VAL  base.calc            IS    10:
VAL  calc.loops           IS    10:
VAL  bundle.size          IS    bundle.size.int TIMES 4:
VAL  packet.size          IS    packet.size.int TIMES 4:

VAL  farmSIZE             IS    MAXnumT8:
VAL  workSIZE             IS    (farmSIZE TIMES 2) - 1:
VAL  data                 IS    1:
VAL  setup                IS    2:
VAL  report               IS    3:
VAL  else                 IS    TRUE:

--------------------------------------------------------------
PROC root(CHAN  OF ANY to.graph, from.graph, requests,
                       results)
--------------------------------------------------------------
  --- internal channels
  CHAN OF ANY   to.handler, from.handler,
                to.router, from.router,
                trigger:

  PAR
    generator(to.router, to.handler, from.handler)
    work.router(to.router, trigger, requests)
    results.router(from.router, trigger, results)
    handler(from.router, to.handler, from.handler, to.graph,
            from.graph)
:
```

```
--------------------------------------------------------
PROC node(CHAN OF ANY    requests.in, requests.out,
                         results.in, results.out,
          VAL INT        proc.id)
--------------------------------------------------------
  --- internal channel declarations
  CHAN OF ANY          to.buffer, to.calculation:
  CHAN OF ANY          from.calculation, from.buffer:
  CHAN OF BOOL         signal:

  PRI PAR
    PAR
      ... request router
      ... request buffer
      ... result buffer
      ... result router
    ... calculation
:
```

```
--------------------------------------------------------
PROC generator(CHAN OF ANY to.router, to.handler,
                           from.handler)
--------------------------------------------------------
  ---                declarations
  [bundle.size]BYTE        bundle:
  INT      tag             RETYPES  [bundle FROM 0 FOR 4]:
  []INT    setup.array     RETYPES  [bundle FROM 4 FOR 8]:
  INT      any:
  SEQ calcs.per.packet = 1 FOR calc.loops
    SEQ
      --- start clock
      to.handler ! calcs.per.packet

      --- initialize nodes
      tag := setup
      to.router ! bundle

      --- generate and send packets
      tag := data
      setup.array[0] := base.calc TIMES calcs.per.packet
      SEQ j = 0 FOR total.bundles
        to.router ! bundle

      --- wait till all results have been received and
          graphed
      from.handler ? any

      --- request report
      tag := report
      to.router ! bundle
:



--------------------------------------------------------
PROC results.router(CHAN OF ANY  from.router, trigger,
                                 results)
--------------------------------------------------------
  --- declarations
  [bundle.size]BYTE  bundle :
  VAL packet.done    IS   TRUE :

  WHILE TRUE
    SEQ
      results ? bundle
      PAR
        trigger ! packet.done
        from.router ! bundle
:
```

```
--------------------------------------------------------------
PROC work.router(CHAN OF ANY to.router, trigger, requests)
--------------------------------------------------------------
  --- declarations
  [bundle.size]BYTE  bundle:
  INT  tag  RETYPES  [bundle FROM 0 FOR 4]:
  BOOL  packet.done, reporting:
  INT   workCOUNT:

  SEQ
    --- initialization
    workCOUNT := 0
    reporting := FALSE

    WHILE TRUE
      PRI ALT
        trigger ? packet.done
          IF
            NOT reporting
              workCOUNT := workCOUNT - 1
            else
              SKIP

        (workCOUNT <= workSIZE) & to.router ? bundle
          IF
            tag = data
              SEQ
                requests ! bundle
                workCOUNT := workCOUNT + 1

            tag = setup
              SEQ
                reporting := FALSE
                requests ! bundle

            tag = report
              SEQ
                reporting := TRUE
                requests ! bundle
:
```

```
--------------------------------------------------------
PROC handler(CHAN OF ANY from.router, to.handler,
             from.handler, screen, keyboard)
--------------------------------------------------------
  --- declarations
  [bundle.size]BYTE       bundle:
  []INT  report.array     RETYPES   [bundle FROM 0 FOR 8]:
  INT    node.id          IS        report.array[0]:
  INT    num.node.bundles IS        report.array[1]:

  VAL    go               IS        1:
  TIMER  clock:
  INT    start.time, stop.time:
  INT    calcs.per.packet:

  SEQ
    WHILE TRUE
      SEQ
        --- start clock on controls command
        to.handler ? calcs.per.packet
        clock ? start.time

        --- receive data packets
        SEQ i = 0 FOR total.bundles
          from.router ? bundle

        --- stop the timer
        clock ? stop.time

        --- let controller know all done graphing
        from.handler ! go

        --- make terminal report
        write.int(screen,(stop.time-start.time) TIMES 64,9)
        write.int(screen,(calcs.per.packet TIMES
                                          base.calc), 4)
        SEQ i = 0 FOR farmSIZE
          SEQ
            from.router ? bundle
            write.int(screen, num.node.bundles, 4)
            newline(screen)

:
```

```
-------------------------------------------------------------
                        request router
-------------------------------------------------------------
--- declarations
[bundle.size]BYTE bundle:
INT tag RETYPES   [bundle FROM 0 FOR 4]:
BOOL d.buffer.empty :

SEQ
  d.buffer.empty := TRUE
  WHILE TRUE
    PRI ALT
      signal ? d.buffer.empty
        SKIP
      requests.in ? bundle
        IF
          tag = data
            IF
              d.buffer.empty
                SEQ
                  d.buffer.empty := FALSE
                  to.buffer ! bundle
              else
                requests.out ! bundle

          tag = setup
            IF
              proc.id < (MAXnumT8-1)
                PAR
                  requests.out    ! bundle
                  to.buffer ! bundle

              else   --- last node
                to.buffer ! bundle

          tag = report
            IF
              proc.id < (MAXnumT8-1)
                PAR
                  requests.out    ! bundle
                  to.buffer ! bundle
              else
                to.buffer ! bundle
```

```
-------------------------------------------------------------
                        request buffer
-------------------------------------------------------------
--- declarations
[bundle.size]BYTE bundle:
INT tag RETYPES   [bundle FROM 0 FOR 4]:
VAL buffer.empty  IS TRUE :

WHILE TRUE
  SEQ
    to.buffer ? bundle
      IF
        tag = data
          SEQ
            to.calculation ! bundle
            signal ! buffer.empty

        tag = setup
          to.calculation ! bundle
        tag = report
          to.calculation ! bundle



-------------------------------------------------------------
                        result buffer
-------------------------------------------------------------
[bundle.size]BYTE  bundle:
WHILE TRUE
  SEQ
    from.calculation ? bundle
    from.buffer ! bundle



-------------------------------------------------------------
                        result router
-------------------------------------------------------------
[bundle.size]BYTE  bundle:
WHILE TRUE
  PRI ALT
    from.buffer ? bundle
      results.out ! bundle
    results.in    ? bundle
      results.out ! bundle
```

```
-------------------------------------------------------------
                        calculation
-------------------------------------------------------------
--- declarations
[bundle.size]BYTE    bundle.in, bundle.out:
[]INT setup.array    RETYPES [bundle.in FROM 4 FOR 8]:
[]INT report.array   RETYPES [bundle.out FROM 0 FOR 8]:
[packet.size]BYTE    work.array:
INT tag RETYPES      [bundle.in FROM 0 FOR 4]:
INT calcs.per.packet IS setup.array[0]:
INT num.node.bundles:
REAL32 x:

SEQ
  WHILE TRUE
    SEQ
      to.calculation ? bundle.in
      IF
        tag = data
          SEQ
            SEQ i = 1 FOR packets.per.bundle
              SEQ
                [work.array FROM 0 FOR 4] := [bundle.in FROM
                      (i TIMES 4) FOR 4]
                SEQ j = 0 FOR calcs.per.packet
                  x := x*x
                SEQ j = 0 FOR packet.size
                  bundle.out[j] := 2(BYTE)

            from.calculation ! bundle.out
            num.node.bundles := num.node.bundles + 1

        tag = setup
          SEQ
            num.node.bundles := 0
            x := 0.9999(REAL32)

        tag = report
          SEQ
            report.array[0] := proc.id
            report.array[1] := num.node.bundles
            from.calculation ! bundle.out
```

APPENDIX C

COORDINATE TRANSFORMATION PROBLEM
SOURCE CODE

Only those portions of code different from that of the generic workfarm are included in this Appendix.

```
--- global variable file
VAL  total.bundles          IS    128:
VAL  total.packets          IS    16384:
VAL  packets.per.bundle     IS    total.packets/total.bundles:
VAL  bundle.size            IS    385:


------------------------------------------------------------
PROC generator(CHAN OF ANY to.router, to.handler,
               from.handler, from.scanner)
------------------------------------------------------------
  --- declarations
  [bundle.size]BYTE   bundle:
  []REAL32 setup.array RETYPES [bundle FROM 4 FOR 24]:
  BYTE   tag        IS bundle[0]:
  REAL32 ASV.pitch  IS setup.array[0]:
  REAL32 ASV.az     IS setup.array[1]:
  REAL32 ASV.roll   IS setup.array[2]:
  REAL32 ASV.x      IS setup.array[3]:
  REAL32 ASV.y      IS setup.array[4]:
  REAL32 ASV.z      IS setup.array[5]:
  INT    index, any:

  SEQ
    --- start clock
    to.handler ! 1

    --- initialize nodes
    tag := setup
    ASV.pitch   :=   0.0(REAL32) ---asv.pitch
    ASV.az      :=   0.0(REAL32) ---asv.az
    ASV.roll    :=   0.0(REAL32) ---asv.roll
    ASV.x       :=   0.0(REAL32) ---asv.x
    ASV.y       :=   64.0(REAL32) ---asv.y
    ASV.z       :=   -8.0(REAL32) ---asv.z
    to.router ! bundle
```

```
    --- generate and send packets
    tag := data
    SEQ i = 0 FOR 128
      SEQ
        SEQ j = 0 FOR 128
          SEQ
            index := j TIMES 3
            bundle[index+1] := BYTE i
            bundle[index+2] := BYTE j
            from.scanner ? bundle[index+3]
        to.router ! bundle

    --- request report
    from.handler ? any
    tag := report
    to.router ! bundle
:


--------------------------------------------------------------
PROC handler(CHAN  OF ANY from.router, to.handler,
             from.handler, to.graph, from.graph)
--------------------------------------------------------------
  --- declarations
  [128][128]INT   terrain.map:
  [bundle.size]BYTE   bundle:
  []INT   report.array   RETYPES   [bundle FROM 0 FOR 8]:
  INT     node.id        IS        report.array[0]:
  INT     node.bundles   IS        report.array[1]:
  VAL     ready          IS        1:
  INT     start.time, stop.time :
  INT     total.bundles:
  TIMER   clock:
  INT     x.int, y.int, z.int:
  INT     any, index:

  SEQ
    --- init
    SEQ i = 0 FOR 128
      SEQ j = 0 FOR 128
        terrain.map[i][j] := 0

    WHILE TRUE
      SEQ
        --- start clock on controls command
        to.handler ? any
        clock ? start.time
```

```
--- receive data packets
SEQ i = 0 FOR 128
  SEQ
    from.router ? bundle
    SEQ j = 0 FOR 128
      SEQ
        index := j TIMES 3
        x.int := INT bundle[index]
        y.int := INT bundle[index+1]
        z.int := (INT bundle[index+2]) - 128
        terrain.map[y.int][x.int] := z.int

--- stop the timer
clock ? stop.time

--- let controller know all done graphing
from.handler ! ready

--- make terminal report
newline(to.graph)
newline(to.graph)
write.int(to.graph,(stop.time-start.time)*64,10)
newline(to.graph)
total.bundles := 0
SEQ i = 0 FOR farmSIZE
  SEQ
    from.router ? bundle
    write.int(to.graph, node.id, 3)
    write.int(to.graph, node.bundles, 10)
    total.bundles := total.bundles + node.bundles
    newline(to.graph)
newline(to.graph)
write.int(to.graph, total.bundles, 13)

--- display wire terrain graph on SONY
[128]INT altitude.array:
VAL  XMID              IS       256:
VAL  YBASE             IS       450:
VAL  scalefactor       IS       200000/256:
INT  reply, horiz, vert, x.old, y.old, x.new, y.new:
INT  horiz1, vert1:
```

```
SEQ
  --- initialization
  SEQ i = 0 FOR 128
    altitude.array[i] := 512
  vert := 0
  to.graph ! c.select.bg.colour;24
  from.graph ? reply
  to.graph ! c.select.fg.colour;2
  from.graph ? reply
  to.graph ! c.init.crt
  from.graph ? reply
  to.graph ! c.select.screen;0
  from.graph ? reply
  to.graph ! c.clear.screen;0
  from.graph ? reply
  to.graph ! c.display.screen;0
  from.graph ? reply
  to.graph ! c.move;256;450
  from.graph ? reply

  SEQ i = 0 FOR 128
    SEQ
      horiz := 2
      horiz1 := (horiz * scalefactor)/1000
      vert1 := (vert * scalefactor)/1000
      x.old := XMID - vert
      y.old := YBASE - (vert1 + terrain.map[i][0])

      --- process row
      SEQ j = 1 FOR 127
        SEQ
          x.new := (XMID + horiz) - vert
          y.new := YBASE - (horiz1 + (vert1 +
                    terrain.map[i][j]))
          --- drawline
          IF
            y.new < altitude.array[j]
              SEQ
                to.graph !  c.draw.line; x.old;
                     y.old; x.new; y.new
                from.graph ? reply
                altitude.array[j] := y.new
            else
              SKIP
          x.old := x.new
          y.old := y.new
          horiz := horiz + 2
          horiz1 := (horiz * scalefactor)/1000
      vert := vert + 2
:
```

```
------------------------------------------------------------
                        Calculation
------------------------------------------------------------

--- bundle declarations
[bundle.size]BYTE     bundle.in:
[bundle.size]BYTE     bundle.out:
[]REAL32 setup.array  RETYPES  [bundle.in FROM 4 FOR 24]:
[]INT    report.array RETYPES  [bundle.out FROM 0 FOR 8]:
BYTE     tag          IS       bundle.in[0]:

INT      node.bundles:

--- computation variables
BYTE   x.byte, y.byte, z.byte:
INT    x.int, y.int, z.int:
REAL32 x.real, y.real, z.real:
REAL32 a,b,c,e,f,g,i,j,k:
REAL32 c4c5, s4c5, d9s8, four, eight, twelve:
REAL32 c4,c5,c6,c7,c8,s4,s5,s6,s7,s8,a7,d9:
REAL32 scanner.az, scanner.pitch, target.range:
REAL32 ASV.x, ASV.y, ASV.z, ASV.az, ASV.pitch, ASV.roll:
REAL32 temp1, temp2, temp3, temp4:
BYTE   raw.scanner.pitch, raw.scanner.az, raw.target.range:

--- constants
VAL Pi     IS 3.1416(REAL32):
VAL PiBy2  IS Pi/2.0(REAL32):
VAL factor IS Pi/180.0(REAL32):

INT  index:
SEQ
  WHILE TRUE
    SEQ
      to.calculation ? bundle.in
      IF
        tag = data
          SEQ
            SEQ z = 0 FOR 128
              SEQ
                index := z TIMES 3
                --- process raw data packet:
                --- pitch into degrees from -15 to -75
                --- convert az into degrees from -40 to +40
                --- convert range into feet from 0 to 32

                temp1 := REAL32 ROUND
                           (INT bundle.in[index+1])
                temp2 := 15.0(REAL32)
                temp3 := 1.47244(REAL32) * temp1
                scanner.pitch := temp1 - (temp2 + temp3)
```

```
        temp4 := REAL32 ROUND
                 (INT bundle.in[index+2])
        scanner.az := (temp4 * 0.6299(REAL32)) -
                       40.0(REAL32)

        target.range := (REAL32 ROUND
              (INT bundle.in[index+3]))/8.0(REAL32)

        --- do sines and cosines
        scanner.pitch :=  (factor * scanner.pitch) -
                           PiBy2
        scanner.az := (factor * scanner.az) - PiBy2
        COSP(c7, scanner.pitch)
        COSP(c8, scanner.az)
        SINP(s7, scanner.pitch)
        SINP(s8, scanner.az)

        --- assign variables
        d9s8    :=    s8*target.range
        four    :=    (d9s8 * c7) + (a7 * c7)
        eight   :=    (d9s8 * s7) + (a7 * s7)
        twelve  :=    c8*target.range

        --- calculate 3 points of the 4x4 matrix
        x.real := ASV.x + ((a * four) + ((b * eight)
                  + (c * twelve)))
        y.real := ASV.y + ((e * four) + ((f * eight)
                  + (g * twelve)))
        z.real := ASV.z + ((i * four) + ((j * eight)
                  + (k * twelve)))

        --- convert results into bytes
        bundle.out[index] := BYTE (INT ROUND x.real)
        bundle.out[index+1] :=  BYTE (INT ROUND
                                      y.real)
        bundle.out[index+2] := BYTE ((INT ROUND
                                    z.real) + 128)
    from.calculation ! bundle.out
    node.bundles := node.bundles + 1

tag = setup
  SEQ
    node.bundles := 0
    ASV.pitch    := setup.array[0]
    ASV.az       := setup.array[1]
    ASV.roll     := setup.array[2]
    ASV.x        := setup.array[3]
    ASV.y        := setup.array[4]
    ASV.z        := setup.array[5]
```

```
    --- convert degrees into radians
    ASV.pitch := factor * ASV.pitch
    ASV.az    := factor * ASV.az
    ASV.roll  := factor * ASV.roll

    --- do sines and cosines
    SINP(s4, (ASV.az + Pi))
    SINP(s5, (ASV.pitch - PiBy2))
    SINP(s6, (ASV.roll + Pi))
    COSP(c4, (ASV.az + Pi))
    COSP(c5, (ASV.pitch - PiBy2))
    COSP(c6, (ASV.roll + Pi))

    --- assign variables
    c4c5 := c4*c5
    s4c5 := s4*c5

    a := (c4c5*c6)+(s4*s6)
    b := c4*s5
    c := (c4c5*s6)-(s4*c6)

    e := (s4c5*c6)-(c4*s6)
    f := s4*s5
    g := (s4c5*s6)+(c4*c6)

    i := s5*c6
    j := -c5
    k := s5*s6

    a7 := -0.5(REAL32)

tag = report
  SEQ
    report.array[0]  := proc.id
    report.array[1]  := node.bundles
    from.calculation ! bundle.out
```

```
--------------------------------------------------------
PROC scanner (CHAN OF ANY from.scanner)
--------------------------------------------------------
  --- declarations
  [128][128]BYTE    range:
  VAL               Pi              IS 3.1416(REAL32):
  VAL               factor          IS Pi/180.0(REAL32):
  REAL32            angle.deg, angle.rad, cosangle, deg.inc:
  BYTE              range.byte:

  SEQ
    --- initialize array of range values
    angle.deg := 75.0(REAL32)
    angle.rad := factor * angle.deg
    COSP(cosangle, angle.rad)
    range.byte := BYTE (INT ROUND (64.0(REAL32)/cosangle))
    SEQ i = 0 FOR 128
      SEQ
        SEQ j = 0 FOR 128
          range[i][j] := range.byte
        angle.deg := angle.deg - 0.46875(REAL32)
        angle.rad := factor * angle.deg
        COSP(cosangle, angle.rad)
        range.byte :=  BYTE (INT ROUND
                  (64.0(REAL32)/cosangle))

    --- pump out range BYTEs
    SEQ i = 0 FOR 128
      SEQ j = 0 FOR 128
        from.scanner ! range[i][j]
:
```

APPENDIX D

MANDELBROT SET PROBLEM SOURCE CODE

```
--- Mandelbrot global constants
VAL  MAXnumT8       IS 8:
VAL  packetSIZE     IS 16:
VAL  else           IS TRUE:
VAL  rSIZE          IS 512:
VAL  iSIZE          IS 512:
VAL  rSTEPS         IS rSIZE/packetSIZE:
VAL  packetCOUNT    IS rSTEPS * iSIZE :
VAL  countLIMIT     IS 255:

VAL  farmSIZE       IS MAXnumT8:
VAL  workSIZE       IS (farmSIZE * 2) - 1 :

VAL  data           IS 1:
VAL  setup          IS 2:
VAL  report         IS 3:


--------------------------------------------------------
PROC root(CHAN OF ANY to.graph, from.graph,
                      requests, results)
--------------------------------------------------------
  --  internal channels
  CHAN OF ANY         to.router, from.router:
  CHAN OF ANY         to.handler, from.handler:
  CHAN OF ANY         trigger:

  VAL  zoom.in     IS 0:
  VAL  zoom.out    IS 1:
  VAL  rSIZE.real IS (REAL64 ROUND rSIZE):
  VAL  iSIZE.real IS (REAL64 ROUND iSIZE):

PAR
  generator(to.router, to.handler, from.handler)
  work.router(to.router, trigger, requests)
  results.router(from.router, trigger, results)
  handler(from.router, to.handler,  from.handler, to.graph,
        from.graph)
:
```

```
--------------------------------------------------------
PROC generator(CHAN OF ANY to.router, to.handler,
                            from.handler)
--------------------------------------------------------
  --  data variables
  [12]BYTE data.array:
  INT tag RETYPES [data.array FROM 0 FOR 4]:
  []INT START RETYPES  [data.array FROM 4 FOR 8]:
  INT rSTART IS START[0]:
  INT iSTART IS START[1]:
  REAL64 setup.value RETYPES [data.array FROM 4 FOR 8]:
  REAL64     rMIN, rMAX, iMIN, iMAX :
  REAL64     rMIN.temp, rMAX.temp, iMIN.temp, iMAX.temp :
  REAL64     ul.x.real,ul.y.real,lr.x.real,lr.y.real:
  REAL64     rMID, iMID:
  INT        mode :
  INT        any, ul.x, ul.y, lr.x, lr.y :

  SEQ
    rMIN := -2.0(REAL64)
    rMAX :=  0.5(REAL64)
    iMIN :=  1.25(REAL64)
    iMAX := -1.25(REAL64)
    WHILE TRUE
      SEQ
        --- initialize nodes
        tag := setup
        setup.value := rMIN
        to.router   !  data.array
        setup.value := rMAX
        to.router   !  data.array
        setup.value := iMIN
        to.router   !  data.array
        setup.value := iMAX
        to.router   !  data.array

        to.handler ! 1

        --- send packets
        tag := data
        SEQ i = 0 FOR iSIZE
          SEQ
            iSTART := i
            SEQ j = 0 FOR rSTEPS
              SEQ
                rSTART := j * packetSIZE
                to.router ! data.array

        --- report
        from.handler ? any
        tag := report
        to.router ! data.array
```

```
--- get new plot coordinates
from.handler ? mode; ul.x; ul.y; lr.x; lr.y
INT32TOREAL64(ul.x.real,ul.x)
INT32TOREAL64(ul.y.real,ul.y)
INT32TOREAL64(lr.x.real,lr.x)
INT32TOREAL64(lr.y.real,lr.y)
ul.x.real := REAL64 ROUND ul.x
ul.y.real := REAL64 ROUND ul.y
lr.x.real := REAL64 ROUND lr.x
lr.y.real := REAL64 ROUND lr.y
IF
  mode = zoom.in
    SEQ
      rMIN.temp :=  (((rMAX-rMIN)*ul.x.real)
                        /rSIZE.real)+rMIN
      rMAX.temp := (((rMAX-rMIN)*lr.x.real)
                        /rSIZE.real)+rMIN
      iMIN.temp :=  (((iMIN-iMAX)*lr.y.real)
                        /iSIZE.real)+iMAX
      iMAX.temp := (((iMIN-iMAX)*ul.y.real)
                        /iSIZE.real)+iMAX
      rMIN := rMIN.temp
      rMAX := rMAX.temp
      iMIN := iMIN.temp
      iMAX := iMAX.temp
  mode = zoom.out
    REAL64 scale.factor:
    SEQ
      scale.factor :=  rSIZE.real/
                    (lr.x.real-ul.x.real)
      rMID := (rMAX + rMIN)/(2.0(REAL64))
      iMID := (iMAX + iMIN)/(2.0(REAL64))
      rMIN := rMID - (scale.factor*(rMID-rMIN))
      rMAX := rMID + (scale.factor*(rMAX-rMID))
      iMIN := iMID - (scale.factor*(iMID-iMIN))
      iMAX := iMID + (scale.factor*(iMAX-iMID))
:
```

```
--------------------------------------------------------------
PROC handler(CHAN OF ANY from.router, to.handler,
                from.handler, to.graph, from.graph)
--------------------------------------------------------------
  --- declarations
  [16 + packetSIZE]BYTE graph.array :
  INT v.c.man RETYPES [graph.array FROM 0 FOR 4]:
  INT v.pSIZE RETYPES [graph.array FROM 4 FOR 4]:
  []BYTE result.array  IS [graph.array FROM 8 FOR
                              (8+packetSIZE)]:
  INT node.packets RETYPES [result.array FROM 0 FOR 4]:
  INT node.loops   RETYPES [result.array FROM 4 FOR 4]:
  BYTE node.id IS result.array[8]:
  INT  range, reply:
  INT  n.x, n.y, m.x, m.y, l.x,l.y, buttons:
  INT  delta.x, delta.y:
  BOOL m.l, m.m, m.r, select.ok, done.select:
  INT  start.time, stop.time, recirc:
  INT  total.loops, total.packets:
  TIMER clock:
  INT any:

  SEQ
    --- initialization
    v.c.man := c.mandelbrot
    v.pSIZE := packetSIZE

    WHILE TRUE
      SEQ
        --- init graphics display
        to.graph   ! c.hide.cursor
        from.graph ? reply
        to.graph   ! c.init.crt
        from.graph ? reply
        to.graph   ! c.select.screen ; 0
        from.graph ? reply
        to.graph   ! c.clear.screen  ; 0
        from.graph ? reply
        to.graph   ! c.display.screen; 0
        from.graph ? reply
        to.graph   ! c.select.colour.table; 1
        from.graph ? reply
        to.graph   ! c.set.colour; countLIMIT; 0; 0; 0
        from.graph ? reply

        --- start clock
        to.handler ? any
        clock ? start.time
```

```
--- receive packets
SEQ i = 0 FOR packetCOUNT
  SEQ
    from.router ? result.array
    to.graph ! graph.array

--- stop clock
clock ? stop.time

--- let controller know all done
from.handler ! 1

--- make terminal report
newline(to.graph)
newline(to.graph)
write.int(to.graph,(stop.time-start.time)*64,10)
newline(to.graph)
total.loops := 0
total.packets := 0
SEQ i = 0 FOR farmSIZE
  BYTE char:
  SEQ
    from.router ? result.array
    write.int(to.graph,(INT node.id),3)
    write.int(to.graph,node.packets,10)
    write.int(to.graph,node.loops,10)
    total.loops := total.loops + node.loops
    total.packets := total.packets + node.packets
    newline(to.graph)
newline(to.graph)
write.int(to.graph, total.packets, 13)
write.int(to.graph, total.loops, 10)
newline(to.graph)
write.int(to.graph,
        (512*(512/packetSIZE))-total.packets,13)
write.int(to.graph,total.loops/total.packets,10)
newline(to.graph)
newline(to.graph)

--- get the new coordinates for calculation
--- set-up for rectangle
to.graph ! c.copy.screen; 0
from.graph ? reply
to.graph ! c.select.fg.colour; 15
from.graph ? reply

--- get the current mouse stats
to.graph ! c.show.cursor
from.graph ? reply
done.select := FALSE
select.ok := FALSE
```

```
WHILE NOT done.select
  SEQ
    --- wait for any mouse button to be pressed
    to.graph ! c.get.mouse
    from.graph ? n.x;n.y;m.l;m.m;m.r
    WHILE (NOT m.m) AND ((NOT m.l) AND (NOT m.r))
      SEQ
        to.graph ! c.get.mouse
        from.graph ? n.x;n.y;m.l;m.m;m.r

    IF
      m.m
        BOOL new.select:
        SEQ
          m.x := n.x
          m.y := n.y
          l.x := m.x
          l.y := m.y
          new.select := TRUE

          --- process mouse input until middle mouse
          --- is released
          WHILE m.m
            SEQ
              to.graph ! c.get.mouse
              from.graph ? n.x;n.y;m.l;m.m;m.r
              IF
                ((n.x <>  l.x) OR (n.y <> l.y)) OR
                                  new.select
                  SEQ
                    new.select := FALSE
                    to.graph ! c.hide.cursor
                    from.graph ? reply
                    to.graph ! c.copy.screen; 1
                    from.graph ? reply

                    --- set new corner coordinates
                    delta.x := n.x - m.x
                    delta.y := n.y - m.y
                    IF
                      delta.x > 0
                        IF
                          delta.y > 0
                            IF
                              delta.x > delta.y
                                delta.x := delta.y
                              TRUE
                                delta.y := delta.x
```

```
                    TRUE
                      IF
                        delta.x > (-delta.y)
                          delta.x := -delta.y
                        TRUE
                          delta.y := -delta.x

                  TRUE
                    IF
                      delta.y > 0
                        IF
                          (-delta.x) > delta.y
                            delta.x := -delta.y
                          TRUE
                            delta.y := -delta.x
                      TRUE
                        IF
                          (-delta.x)>(-delta.y)
                            delta.x := delta.y
                          TRUE
                            delta.y := delta.x
              l.x := n.x
              l.y := n.y
              to.graph ! c.draw.rectangle;
                  m.x; m.y; delta.x; delta.y
              from.graph ? reply
              to.graph ! c.show.cursor
              from.graph ? reply
          TRUE
            SKIP

--- order  the screen coordinates for
--- proper range
l.x := m.x + delta.x
IF
  m.x > l.x
    SEQ
      n.x := m.x
      m.x := l.x
      l.x := n.x
  TRUE
    SKIP
l.y := m.y + delta.y
IF
  m.y > l.y
    SEQ
      n.y := m.y
      m.y := l.y
      l.y := n.y
  TRUE
    SKIP
```

```
            select.ok := (delta.x <> 0) AND
                         (delta.y <> 0)

        --- right mouse button hit, do zoom in
        m.r AND select.ok
          SEQ
            done.select := TRUE
            graph.results ! zoom.in; m.x; m.y;
                                     l.x; l.y

        --- left mouse button hit, do zoom out
        m.l AND select.ok
          SEQ
            done.select := TRUE
            graph.results ! zoom.out; m.x; m.y;
                                      l.x; l.y

        TRUE
          SKIP
:
```

# LIST OF REFERENCES


[At87] Atkin, P., Performance Maximization, INMOS Technical Note Number 17, Bristol, United Kingdom, March 1987.

[Br88] Bryant, G., Design, Implementation, and Evaluation of an Abstract Programming and Communication Interface for a Network of Transputers, M.S. Thesis, U.S. Naval Postgraduate School, Monterey, California, June 1988.

[Ha87] Hart, S.J., Design, Implementation, and Evaluation of a Virtual Shared Memory System in a Multi-Transputer Network, M.S. Thesis, U.S. Naval Postgraduate School, Monterey, California, December 1987.

[HwCh87] Hwang, K., and Chowkwanyun, R., "Dynamic Load Balancing for Distributed Supercomputing and AI Applications," Technical Report CRI-87-04, Computer Research Institute, University of Southern California, Los Angeles, California, January 1987.

[HwGh87] Hwang, K.,and Ghosh, J., "Hypernet: A Communication-Efficient Architecture for Constructing Massively Parallel Computers," IEEE Transactions on Computers, Vol. C-36, No. 12, pp 1450-1466, December 1987.

[In87a] Transputer Reference Manual, INMOS Ltd., Bristol, United Kingdom, January 1987.

[In87b] T800 Preliminary Data Sheet, INMOS Ltd., Bristol, United Kingdom, 1987.

[In88] An Introduction to Transputers, draft 2.0, INMOS Ltd., Colorado Springs, Colorado, January 1988.

[MaSh87] May, D., and Shepherd, R., Communicating Process Computers, INMOS Technical Note Number 22, Bristol, United Kingdom, February 1987.

[Po86] Pountain R., "Turbocharging Mandelbrot," BYTE Magazine, vol.10, no. 9, pp. 359-366, September 1986.

[PoMa87] Pountain, R., and May, D., A Tutorial Introduction to OCCAM Including Language Definition, INMOS Ltdl, Bristol, United Kingdom, March 1987.

[Ri87] Rickenbach, M.D., Correction of Inertial Navigation System Drift Errors For an Autonomous Land Vehicle Using Optical Radar Terrain Data, M.S. Thesis, U.S. Naval Postgraduate School, Monterey, California, September 1987.

[Va87] Vanni, F.J., Test and Evaluation of the Transputer in a Multi-Transputer Configuration, M.S. Thesis, U.S. Naval Postgraduate School, Monterey, California, June 1987.

INITIAL DISTRIBUTION LIST

| | | No. Copies |
|---|---|---|
| 1. | Defense Technical Information Center<br>Cameron Station<br>Alexandria, VA 22304-6145 | 2 |
| 2. | Library, Code 0142<br>Naval Postgraduate School<br>Monterey, CA 93943-5002 | 2 |
| 3. | Director, Information Systems (OP-945)<br>Office of the Chief of Naval Operations<br>Navy Department<br>Washington, D.C. 20350-2000 | 1 |
| 4. | Superintendent, Naval Postgraduate School<br>Computer Technology Programs, Code 37<br>Monterey, CA 93943-5000 | 1 |
| 5. | Department Chairman, Code 52<br>Department of Computer Science<br>Naval Postgraduate School<br>Monterey, CA 93943 | 1 |
| 6. | Major Richard A. Adams, USAF, Code 52<br>Department of Computer Science<br>Naval Postgraduate School<br>Monterey, CA 93943 | 3 |
| 7. | Dr. Uno R. Kodres, Code 52Kr<br>Department of Computer Science<br>Naval Postgraduate School<br>Monterey, CA 93943 | 3 |
| 8. | Daniel Green, Code 20F<br>Naval Surface Weapons Center<br>Dahlgren, VA 22449 | 1 |
| 9. | Jerry Gaston, Code N24<br>Naval Surface Weapons Center<br>Dahlgren, VA 22449 | 1 |
| 10. | Captain J. Hood, USN<br>PMS 400B5<br>Naval Sea Systems Command<br>Washington, D.C. 20362 | 1 |

11. RCA AEGIS Repository 1
RCA Corporation
Government Systems Division
Mail Stop 127-327
Moorestown, NJ 08057

12. Library (Code E33-05) 1
Naval Surface Weapons Center
Dahlgren, VA 22449

13. Dr. M.J. Gralia 1
Applied Physics Laboratory
Johns Hopkins Road
Laurel, MD 20702

14. Dana Small, Code 8242 1
Naval Ocean Systems Center
San Diego, CA 92152

15. Captain W.D. Cloughley, USN (ret) 1
California State University, Sacramento
Technology Transfer Program Office
School of Business & Personal Administration
Room 3057
6000 J Street
Sacramento, CA 95825

16. LT W.R. Cloughley, USN 2
1448 47th Street
Sacramento, CA 95819

17. AEGIS Modelling Laboratory, Code 52 5
Department of Computer Science
Naval Postgraduate School
Monterey, CA 93943

18. LCDR S.J. Hart, RAN 1
59 Yanco Road
West Pymbole, New South Wales
AUSTRALIA

19. LCDR G. Bryant, USN 1
Mare Island Naval Shipyard
Vallejo, CA 94592

Thesis
C5183 Cloughley
c.1 Evaluation of work distribution algorithms and hardware topologies in a multi-Transputer network.